МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННАЯ КОРПОРАЦИЯ «РОССИЙСКАЯ КОРПОРАЦИЯ НАНОТЕХНОЛОГИЙ»

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
"Московский государственный технический университет радиотехники, электроники и автоматики"

Центр проектирования интегральных схем,
устройств наноэлектроники и микросистем

Кафедра физики конденсированного состояния

И.Е. Тарасов, Е.Ф. Певцов

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ СХЕМЫ И ИХ ПРИМЕНЕНИЕ В СХЕМОТЕХНИЧЕСКИХ РЕШЕНИЯХ

Учебное пособие

Москва 2012

ББК 32.844.1 + 32.852
Т 19
УДК 621.3.049.77

Рецензенты: …., к.т.н. Д.С. Потехин.

Т 19 Тарасов И.Е., Певцов Е.Ф. Программируемые логические схемы и их применение в схемотехнических решениях: Учебное пособие / Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Московский государственный технический университет радиотехники, электроники и автоматики" – М., 2012. – 184 с.

В пособии излагаются основы проектирования цифровых систем с использованием языка описания аппаратуры Verilog HDL. Пособие основывается на материалах курса «Системы автоматизированного проектирования» (МГТУ МИРЭА, факультет электроники, 2011 г.).

Учебное пособие содержит базовые сведения для обучения проектированию цифровых систем на основе конструкций языка Verilog. Лабораторные работы практикума ориентированы на среду автоматизированного проектирования Integrated Software Environment (ISE v. 13.2) и помогают получить навыки практической работы, выполнив законченные проекты и реализовав их на демонстрационных платах.

Пособие предназначено для студентов, бакалавров и магистров специальностей 210100, 210104, 210600, 210601, 222900, 210100(550700), а также для студентов других технических специальностей и лиц, занимающихся самообразованием, и имеющих целью получить базовые знания и навыки проектирования систем на основе программируемых логических интегральных схем.

Табл.: 27. Ил.: 117. Библиогр.: 24 назв.

Печатается по решению редакционно-издательского совета университета

ISBN 978-5-7339-0

Тарасов Илья Евгеньевич

Певцов Евгений Филиппович

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ СХЕМЫ И ИХ ПРИМЕНЕНИЕ В СХЕМОТЕХНИЧЕСКИХ РЕШЕНИЯХ

Учебное пособие

Учебное пособие напечатано в авторской редакции

Подписано в печать 00.00.2012. Формат 00x00 0/00.
Усл. печ. л. 0,00. Усл. кр.-отт. 00,00. Уч.-изд. л. 0,0.
Тираж 100 экс. С. 000

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
"Московский государственный технический университет
радиотехники, электроники и автоматики"
119454, Москва, пр-т Вернадского, 78

СОДЕРЖАНИЕ

Введение

Современный маршрут разработки интегральных микросхем предполагает этап макетирования на базе программируемых логических интегральных схем (ПЛИС). Микросхемы такого типа представляют собой матрицу программируемых логических элементов (матрица ПЛМ), между которыми проложены электрически коммутируемые соединения. Это позволяет конфигурировать отдельные компоненты и создавать связи между ними путем загрузки в ПЛИС потока данных, включающего требуемые цепи и узлы коммутации. В результате из имеющихся в составе ПЛМ ресурсов создается требуемая цифровая схема, которая при необходимости может быть легко модифицирована. Многие ПЛИС хранят конфигурацию в статической памяти, которая может быть неограниченно перезаписана, таким образом, единственная программируемая микросхема может служить для макетирования целой серии устройств, для чего кроме самой ПЛИС требуется только компьютер с установленной САПР. Современные ПЛИС имеют достаточно большой объем ресурсов, достигающий миллионов эквивалентных логических вентилей, составляющих сотни тысяч логических ячеек, что позволяют получить макет (прототип) цифрового устройства практически любой сложности. Изготовление соответствующей микросхемы с «жесткими» (реализованными аппаратно, а не с помощью коммутируемых ключей) соединениями сопряжено с существенными материальными затратами, которые, к тому же, увеличиваются с переходом к более современным технологическим процессам.

Таким образом, для проверки функциональных возможностей устройства не требуется проведения полного цикла проектирования топологии кристалла и изготовления. При этом на ранних стадиях проектирования разработчик получает в свое распоряжение аппаратный аналог будущего устройства, который может наглядно продемонстрировать работоспособность принятых проектных решений, что существенно снижает общие временные затраты при проектировании СБИС.

Одновременно весьма существенно падает и стоимость разработки, поскольку однократно приобретаемая ПЛИС используется на протяжении всего цикла разработки, реализуя при этом любое количество модификаций разрабатываемой схемы.

Основой для такого стиля проектирования СБИС является использование специализированных *языков описания аппаратуры* (Hardware Description Languages, HDL), которые позволяют в короткие сроки описать желаемое поведение будущего устройства на языке, приближенном по стилю к языкам программирования. После проверки конструкторских решений в ПЛИС те же файлы с кодом, описывающим поведение устройства, могут быть использованы для получения аналогичной интегральной микросхемы. Использование для проектирования цифровых и смешанных систем языков описания аппаратуры позволяет эффективно использовать ПЛИС на этапе макетирования, переходя впоследствии к изготовлению микросхемы с минимальными изменениями конструкторской документации.

Таким образом, современная идеология проектирования СБИС может быть условно разделена на два основных этапа: логическое поведенческое (back-end) проектирование без привязки к физической реализации и физическое топологическое (front-end) проектирование с размещением на кристалле и привязкой к технологическим процессам изготовления. Логическое проектирование в упрощенном понимании заключается в разработке поведенческой (behavioral) модели на языке HDL и последующем автоматическом преобразовании программного кода в модель устройства, описанную с помощью абстракции *регистровых передач* сигналов (Register Transfer Level, RTL-уровень).

На RTL-уровне поведение схемы определяется в терминах потоков сигналов (или пересылок данных) между аппаратными регистрами и логических операций над данными сигналами. В свою очередь, RTL-описание транслируется специальными программами в проектную документацию для инструментов автоматизированного синтеза. Синтезируемое описание представляет собой файл, в котором схема устройства представлена в виде оптимальных соединений между вентилями, составляющими минимально необходимый полный набор логических элементов. Формат этого файла, называемого EDIF-файлом (от Electronic Distribution International Format), унифицирован, и его содержание воспринимается как исходные данные для всех систем автоматизированного проектирования устройств и систем на основе ПЛИС или СБИС.

В соответствии с базовыми технологиями изготовления СБИС на кристалле или реализации в ПЛИС, EDIF-файл преобразуется в эквивалентное описание схемы в виде таблицы соединений библиотечных элементов (net-лист). На следующих этапах соответствующими инструментами САПР выполняется физическое проектирование: размещение элементов на площади кристалла, трассировка межсоединений, разработка масок для изготовления на фабрике и другие операции маршрута проектирования СБИС.

Результаты практически всех проектных процедур маршрута подлежат верификации (например, проверкам выполнения технологических и проектных норм, требуемых функций, целостности сигналов). Важно отметить, что для процедур верификации также используются HDL-модели устройства и характеристик его библиотечных и паразитных элементов.

В данном пособии подробно излагаются основы проектирования цифровых систем с использованием языка описания аппаратуры **Verilog HDL**. Пособие основывается на материалах курса «Системы автоматизированного проектирования», прочитанного в 2011 г. в Московском государственном техническом университете радиотехники, электроники и автоматики на факультете электроники.

Первый раздел содержит общие сведения о структуре программируемых интегральных схем, а также основные технические характеристики ПЛИС семейств Spartan 3 и Spartan 6.

Во втором разделе на примере простого устройства комбинационной логики подробно рассмотрены основные процедуры выполнения проекта в среде автоматизированного проектирования **Integrated Software Environment** (v. 13.2) фирмы XILINX [1.1]. Аппаратная реализация проекта выполнена на базе микросхемы FPGA 6SLX45CSG324 семейства Spartan-6, установленной на демонстрационной плате ATLYS (http://www.digilentic.com).

Третий раздел посвящен изложению основ языка описания аппаратуры HDL Verilog. Приведены сведения по синтаксису и основным конструкциям языка.

В четвертом разделе содержатся примеры описания устройств комбинационной и последовательностной логики на языке Verilog.

В пятом разделе, основанным на учебных материалах компании Xilinx [1], приводятся задания лабораторного практикума по основам Verilog, которые построены таким образом, чтобы помочь обучающимся на практике преодолеть психологический барьер для перехода к практической работы с микросхемами ПЛИС. Для этой цели рекомендуется выполнять учебные задания, доводя их до прошивки в ПЛИС, установленные на демонстрационных платах ALTYS или Spartan 3E (производство компании Digilent [2]).

1. Архитектура ПЛИС фирмы Xilinx

1.1 Общие сведения о ПЛИС

Программируемые интегральные логические схемы предоставляют возможности быстрого создания цифровых устройств с задаваемой пользователем внутренней структурой. Изменение принципиальной электрической схемы СБИС в кристалле ПЛИС выполняется путем перепрограммирования. В результате цикл создания сложных цифровых устройств, включая разработку многопроцессорных систем и систем параллельной обработки больших массивов данных с накоплением, может составлять буквально несколько дней. Значительно сокращается и стоимость всего проекта в целом, т.к. СБИС ПЛИС с топологическими нормами до 45 нм выпускаются серийно и вполне доступны на рынке.

В настоящее время наиболее распространены ПЛИС с двумя разными архитектурами:

1) **CPLD** (**Complex Programmable Logic Device**) – устройства, использующие для хранения конфигурации энергонезависимую память (Flash или EEPROM);

2) **FPGA** (**Field Programmable Gate Array**) – устройства, использующие для хранения конфигурации энергозависимую память, загружаемую от специального теневого ПЗУ после включения питания.

Микросхема CPLD состоит из матрицы однотипных логических макроячеек (см. [1.2…1.4] и рисунок 1.1).

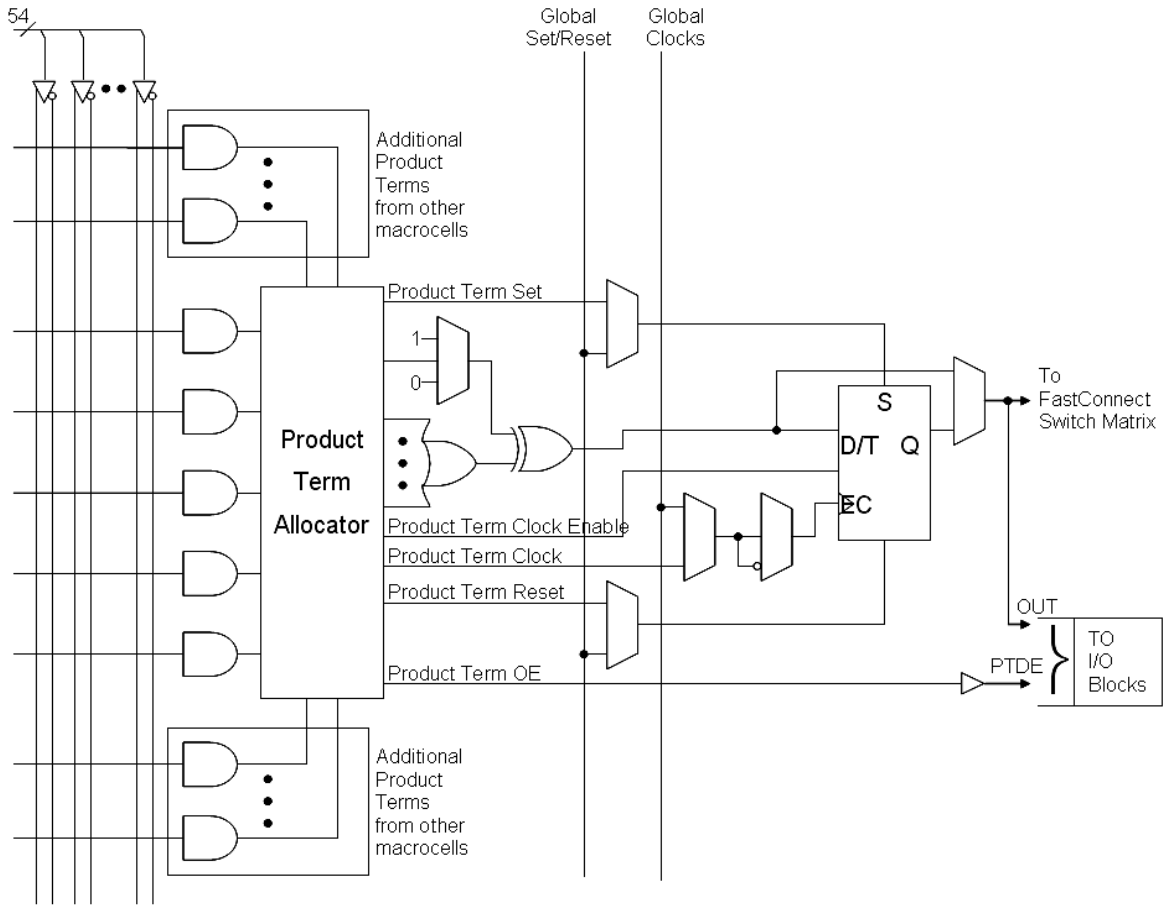


Рисунок 1.1 - Типовая макроячейка ПЛИС CPLD Xilinx.

В каждой макроячейке имеется D-триггер и распределитель термов (Product Term Allocator), способный синтезировать логическую функцию 54-х (серии XC9500XL и XC9500VX) или 36-ти (XC9500) переменных. Такая внутренняя структура макроячеек и комбинированные перепрограммируемые соединения нескольких макроячеек позволяют реализовать на их базе разнообразные цифровые модули и схемные решения.

Соединения макроячеек, образующих функциональные блоки, буферных блоков ввода-вывода (Input/Output block, IOB) и подключение внешних выводов реализуются с помощью входящих в состав кристалла специальных трассировочных логических устройств (матриц Fast Connect). Структурная схема микросхемы ПЛИС семейства CDLD XC9500XL изображена на рисунке 1.2.

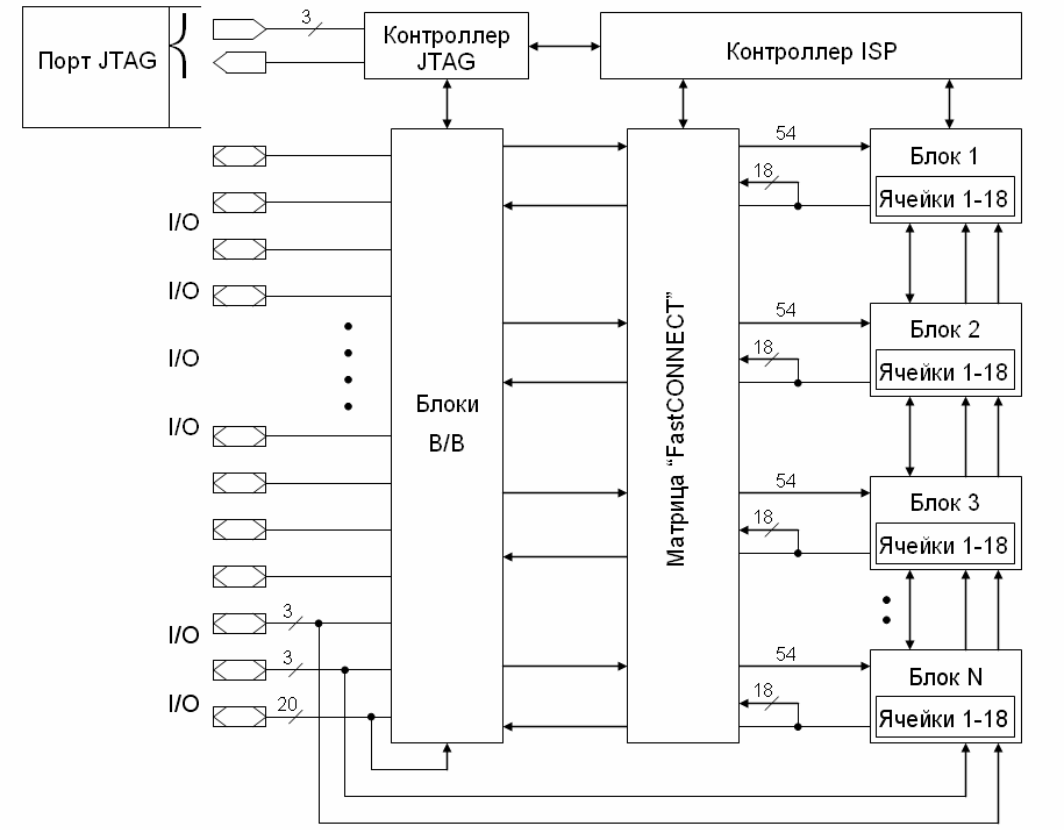


Рисунок 1.2 - Структурная схема CPLD XC9500XL.

Дополнительно на кристалле могут быть размещены специальные элементы, позволяющие организовать синхронную работу схемы, обеспечивающие высокие показатели по быстродействию (блок Global Clock или GCK), а также элементы, выполняющие начальную установку (Global Set/Reset, GSR) и перевод выходных буферов в состояние с высоким импедансом (Global Tri-State, GTS). Контроллеры стандартного интерфейса программирования конфигурационной памяти JTAG и флеш-ПЗУ также расположены на кристалле, что обеспечивает возможность перепрограммирования ПЛИС CPLD непосредственно в системе (in-system programmable) с помощью специальных выделенных выводов для перепрограммирования микросхемы.

Основной интерес при макетировании и разработке цифровых устройств общего назначения представляют ПЛИС с архитектурой FPGA (field-programmable gate array), которые предоставляют наиболее широкие функциональные возможности и наибольшее количество аппаратных ресурсов. Микросхемы FPGA используют энергозависи-

мую статическую память, и каждый раз в начале работы требуют процедуры загрузки конфигурационной информации.

Основным программируемым ресурсом является так называемая логическая ячейка, состоящая из генератора логических функций, работа которого задается таблицей истинности (**Look-Up Table** или **LUT**), триггера и некоторого количества специализированных ресурсов, облегчающих реализацию типичных для цифровой схемотехники узлов. Логические ячейки образуют прямоугольную матрицу, окруженную блоками ввода-вывода, обеспечивающими подключение внутренних линий к внешним выводам корпуса ПЛИС. Внутри матрицы логических ячеек и между этой матрицей и блоками ввода-вывода находятся трассировочные линии – программируемые электрические соединения. ПЛИС может реализовать практически любое цифровое устройство путем программирования таблиц истинности отдельных ячеек и управляемой коммутации соответствующих трассировочных ресурсов.

Логические ячейки объединяются в *конфигурируемый логический блок* (КЛБ) (Configurable Logic Block, CLB). Как видно из рисунка 1.3, в состав CLB входят два D-триггера и два генератора логических функций LUT, каждый из которых имеет четыре входа и образует, таким образом, базовый блок статической памяти с организацией 16x1. Это дает возможность использовать в проектах блоки распределенной памяти с синхронным или асинхронным интерфейсом и двумя портами.

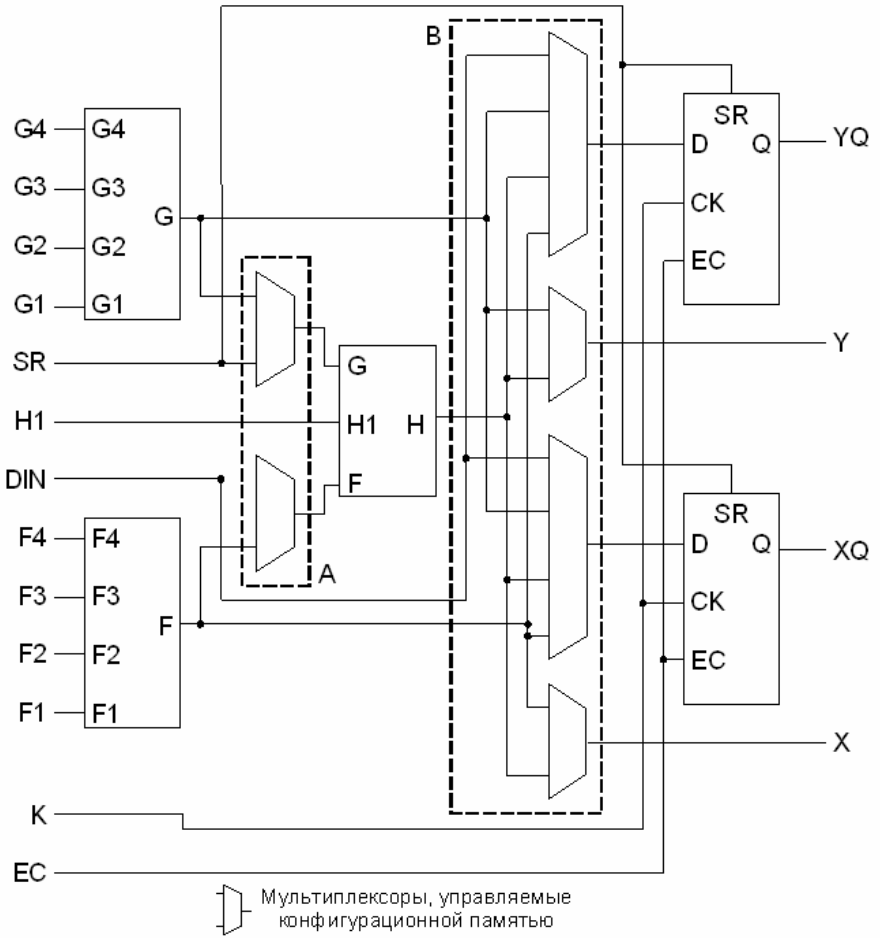


Рисунок 1.3 - Структурная схема базового блока FPGA семейства микросхем Spartan, выпускаемых фирмой Xilinx.

Типовая архитектура микросхем ПЛИС, выполненная по технологии FPGA изображена на рисунке 1.4.

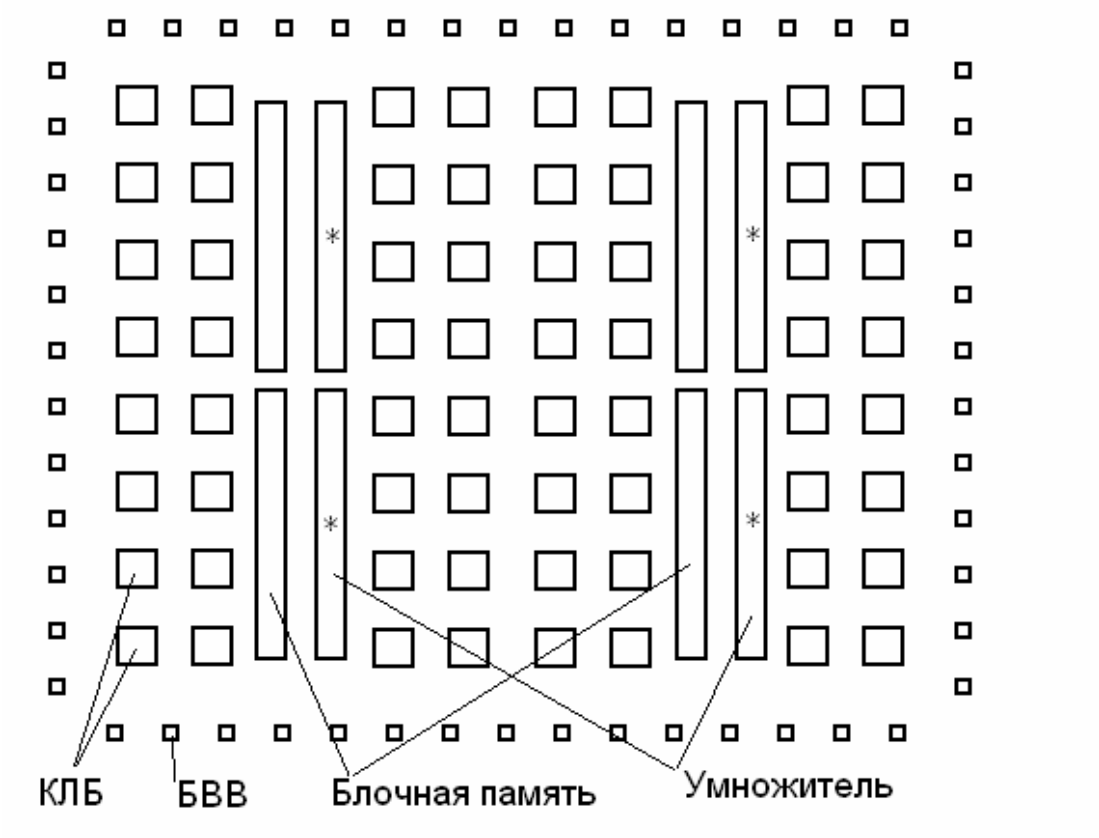


Рисунок 1.4 - Организация ПЛИС с архитектурой FPGA.

В ПЛИС FPGA матрица конфигурируемых логических блоков CLB, окружена по периферии кристалла интерфейсными блоками IOB, которые соединены с внешними выводами. Между CLB и IOB посредством программируемых трассировочных линий (Routing Channels) формируются необходимые межсоединения. В результате затраты на трассировочные индивидуальные соединения каждого функционального блока с внешними выводами существенно снижаются, и в FPGA микросхемах удается разместить до десятков тысяч CLB.

В ПЛИС FPGA фирмы Xilinx на кристалле размещены следующие ресурсы:

1) Модули формирования тактового сигнала.

Данные модули представляют собой аппаратные устройства, способные сформировать тактовый сигнал с характеристиками, требуемыми для высокоскоростной цифровой электроники. В ПЛИС Spartan-3 такие модули обозначаются как DCM (*Digital Clock Manager*) и основаны на устройстве с цифровой автоподстройкой задержки (DLL, *Delay-Locked Loop*). Выходы DCM подключаются к глобальным тактовым цепям, предназначенным для подачи тактового сигнала одновременно на все компоненты ПЛИС, что обеспечивает ее стабильную работу. Использование DCM настоятельно рекомендуется во всех проектах.

2) Цепи ускоренного переноса.

Эти цепи предназначены для быстрой передачи бита переноса между логическими ячейками, что позволяет достаточно просто организовывать многоразрядные уз-

лы. Такой подход имеет два основных преимущества. Прежде всего, выделенные цепи ускоренного переноса обладают меньшими задержками по сравнению с прочими трассировочными ресурсами. Другим преимуществом является высвобождение той части трассировочных ресурсов, которые в противном случае оказались бы задействованными для организации каскадного соединения ячеек. Однако это свойство играет меньшую роль по сравнению с существенным увеличением тактовой частоты, достигаемой для многоразрядных устройств, таких как счетчики, сумматоры и т.п. Усовершенствование цепей ускоренного переноса является постоянной тенденцией развития FPGA.

3) Блочная память.

Ресурс этого типа появился в ПЛИС сравнительно недавно, и по своей сути характерен именно для устройств с архитектурой FPGA. Иерархия памяти для этих ПЛИС выглядит следующим образом. Наиболее быстрым, и в то же время ресурсоемким способом организации памяти является использование триггеров логических ячеек. Очевидно, что при таком подходе каждая ячейка может хранить только один бит данных, и организация сколько-нибудь существенных массивов памяти весьма затруднена. Отличительной особенностью архитектуры устройств, предлагаемых фирмой Xilinx, являлась возможность конфигурирования логических генераторов в качестве устройств статической памяти с организацией 16x1 (т.н. распределенная память). В результате мкость того же кристалла увеличивается ровно в 16 раз. Распределенная память может быть полезна в целом ряде случаев, однако организация крупных массивов наталкивается на существенное ограничение, связанное с необходимостью трассировки по кристаллу многочисленных линий адреса и данных. Производительность проекта, использующего ПЛИС FPGA в качестве большого модуля ОЗУ, оказывается в таком случае весьма невысокой. Кардинальным решением данной проблемы явилось размещение на кристалле FPGA блоков синхронной двупортовой статической памяти, работающей на достаточно высокой частоте. Первоначально блоки имели размер 4 кбит (с возможностью использования различных вариантов разрядности), в последних же семействах введены 18-кбитные блоки. Подобное решение позволяет реализовывать достаточно гибкие и универсальные схемы взаимодействия вычислительных устройств и накристальной памяти.

4) Выделенные умножители.

На определенном этапе развития ПЛИС выяснилось, что программируемая логика может рассматриваться не только в качестве экзотического решения для прототипирования новых устройств, но и в качестве эффективного сопроцессора цифровой обработки сигналов. Возможность реализации на кристалле ПЛИС параллельных вычислений существенно улучшает их позицию по сравнению с широко распространенными сигнальными процессорами. При определенных условиях соотношение «производительность/цена» для ПЛИС существенно выше, чем для таких процессоров. Эти условия реализуются в основном в тех случаях, когда решение задачи требует выполнения большого количества операций вида «умножение с накоплением», характерных для цифровой фильтрации, спектрального анализа, нейросетевых алгоритмов и т.п. Поэтому для повышения эффективности ПЛИС в этих задачах на кристалл последовательно вводились устройства, облегчающие построение умножителей независимых операндов, вплоть до введения на кристалл специализированных блоков, выполняющих умножение независимых 18-битных операндов на частотах до сотен мегагерц (266 в рассматриваемом семействе Spartan-3E). На данном этапе можно говорить, что использование ПЛИС в качестве устройств DSP вполне оправдано не только с технической, но и с экономической точки зрения.

1.2 Основные характеристики семейства Spartan-3

До недавнего времени наиболее современным семейством недорогих ПЛИС с архитектурой FPGA фирмы Xilinx являлось Spartan-3, изготовленное по 90-нм техпроцессу. Оно имеет несколько модификаций, последовательно появлявшихся с момента анонсирования первой разновидности в 2004 году. Впоследствии были выпущены подсемейства Spartan-3E, Spartan-3A, Spartan-3A DSP, Spartan-3AN.

Каждый КЛБ семейства Spartan-3E состоит из четырех секций, сгруппированных в пары (рисунок 1.5). Левая пара называется SLICEM и содержит полнофункциональные логические генераторы, которые могут использоваться также в качестве распределенного ОЗУ или сдвигового регистра. Однако расположенная на рисунке справа пара SLICEL может реализовать только логику.

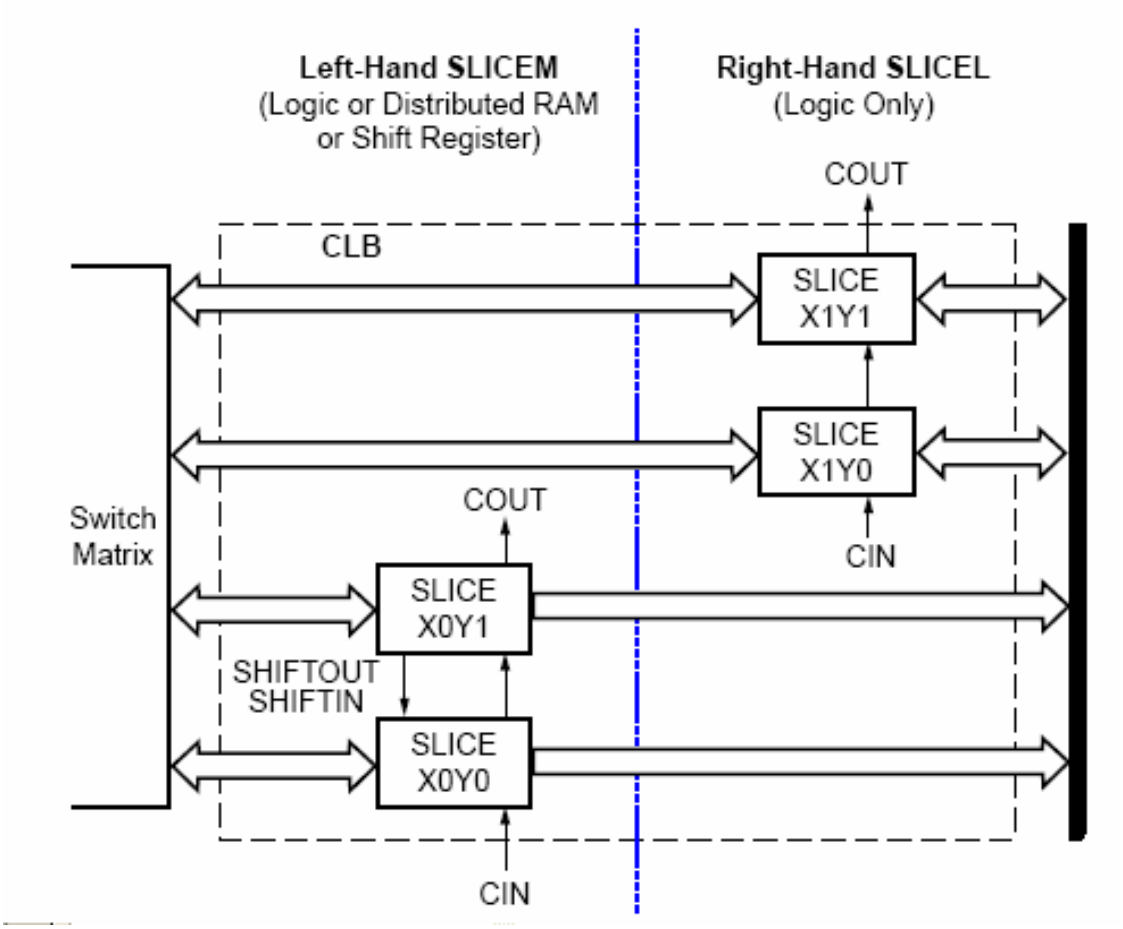


Рисунок 1.5 - Организация секций КЛБ в ПЛИС Spartan-3E.

Фирма Xilinx уделяет достаточно большое внимание соблюдению баланса между ресурсами, размещаемыми на кристалле ПЛИС. Одним из основных моментов, относящихся к структуре КЛБ, является оценка процентного соотношения распределенной памяти и собственно логических ресурсов. Собственные оценки логической емкости ПЛИС самой фирмой Xilinx часто приводятся из соотношения «25% ячеек используется в качестве распределенной памяти, остальное – в качестве логики». Таким образом, по оценке Xilinx, в среднем около четверти ячеек используется «не по прямому назначению», т.е. в качестве распределенной памяти или сдвиговых регистров. Можно предположить, что большая часть логических ячеек скорее всего будет реализовывать именно логические функции, а уменьшение максимального объема доступной распре-

деленной памяти прекрасно компенсируется увеличением количества блочной (гораздо более эффективной и удобной в использовании). Наконец, следует все-таки отметить, что распределенная память в секциях SLICEM так и осталась распределенной по всему кристаллу. В итоге полнофункциональные SLICEM и облегченные SLICEL вполне могут дополнять друг друга.

Упрощенная схема компонентов отдельных секций показана на рисунке 1.6.

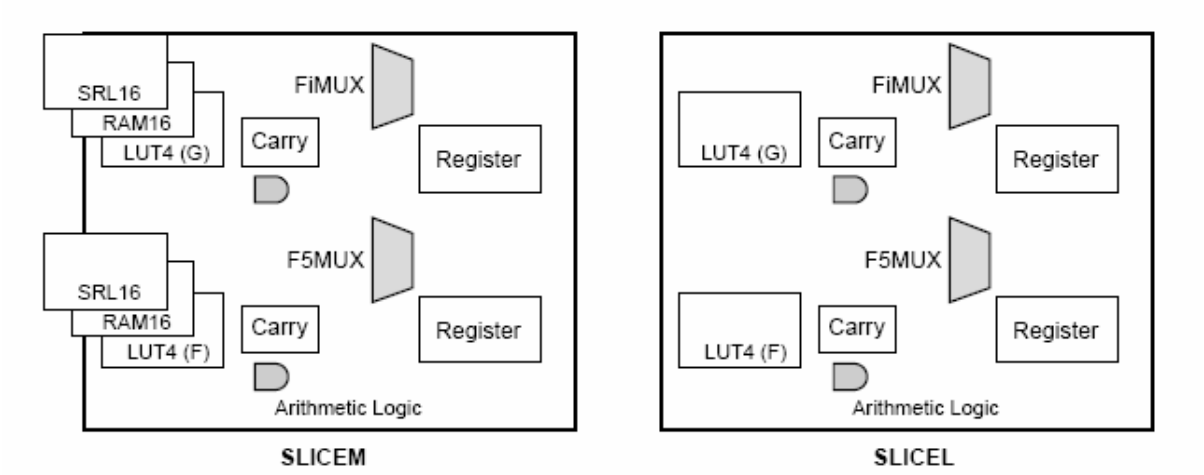


Рисунок 1.6 Упрощенное графическое изображение компонентов секций семейства Spartan-3E.

Технические характеристики ПЛИС Spartan3 и Spartan-3E приведены в таблице 1.1.

Таблица 1.1 – Основные технические характеристики ПЛИС Spartan3 и Spartan-3E.

ПЛИС	Логические ячейки	Системные вентили	Распределенная память, бит	Блоков умножения	Максимальное число программируемых выводов
Spartan-3 – 1,2 В FPGA с выделенными умножителями					
XC3S50	1728	50 тыс.	12 К	–	124
XC3S200	4320	200 тыс.	30 К	12	173
XC3S400	8064	400 тыс.	56 К	16	264
XC3S1000	17280	1 млн.	120 К	24	391
XC3S1500	29952	1,5 млн.	208 К	32	487
XC3S2000	46080	2 млн.	320 К	40	565
XC3S4000	62208	4 млн.	432 К	96	712
XC3S5000	74880	5 млн.	520 К	104	784
Spartan-3E – 1,2 В FPGA с выделенными умножителями					
XC3S100E	2160	100 тыс.	15 К	4	108
XC3S250E	5508	250 тыс.	38 К	12	172
XC3S500E	10476	500 тыс.	73 К	20	232
XC3S1200E	19512	1,2 млн.	136 К	28	304
XC3S1600E	33192	1,6 млн.	231 К	36	376

1.3 Основные характеристики семейства Spartan-6

Упрощенная архитектура ПЛИС нового семейства Spartan-6 приведена на рисунке 1.7. Эти микросхемы продолжают линию недорогих ПЛИС с архитектурой FPGA, выпускаемых фирмой Xilinx.

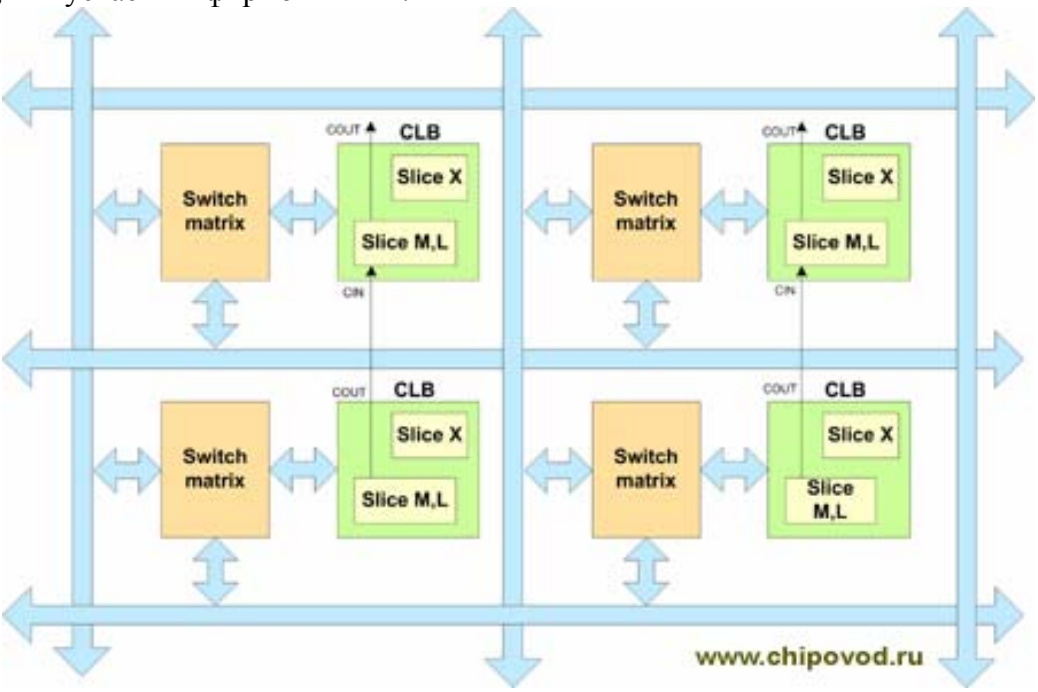


Рисунок 1.7 - Архитектура FPGA Spartan-6.

Каждый КЛБ семейства подключен к трассировочной матрице, и имеет цепь переноса от соседнего КЛБ и содержит две секции Slice: один SLICEX и один SLICEM или SLICEL (см. рисунок 1.8).

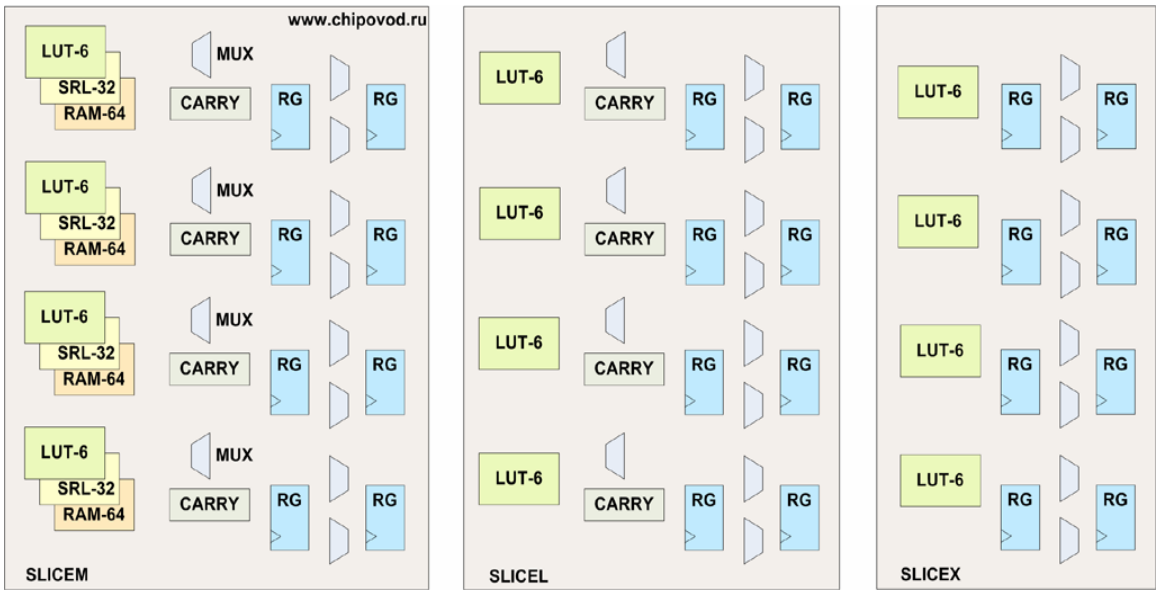


Рисунок 1.8 - Упрощенное графическое изображение компонентов секций семейства Spartan-6.

Наиболее сложная секция **SLICEM** содержит следующие компоненты:

- 4 таблицы преобразования LUT- 6, каждая из которых имеет шесть входов и может конфигурироваться как 4 регистра по 32-разряда, которые могут использоваться как сдвиговые регистры;
- 4 блока распределенной памяти (distributed RAM) 64бит x 1 =256 бит;
- цепи быстрого переноса CARRY, которые нужны в том числе для построения сумматоров;
- расширяемые мультиплексоры MUX;
- 8 регистров для хранения результата.

LUT КЛБ могут реализовать любую булевую функцию 6-ти переменных. Мультиплексоры-расширители MUX позволяют объединять выходы таблиц LUT для получения функций 7 и 8 переменных. Кроме этого имеется 8 триггеров, 4 из которых конфигурируются как D-триггеры для хранения результата таблиц LUT или как latch–триггеры, переключающиеся по уровню тактового сигнала. В отличие от семейства Spartan-3 имеется 4 дополнительных триггера, для хранения промежуточного результата. Сдвиговые регистры SRL-32 являются 32-х разрядными. Так называемая распределенная память (distributed RAM), реализована в виде блоков по 64 бит, поэтому с одной секции **SLICE** можно получить до 256 бит памяти. Такая память удобна для маленьких буферов или регистровых файлов, если необходим больший объем – необходимо использовать блочную память (Block RAM) объем которой в семействе Spartan-6 увеличен (см. таблицу 1.2).

Таблица 1.2 – Сравнительные характеристики блочной памяти семейств Spartan.

	Spartan-6	Spartan-3	Spartan-3A DSP
Число блоков	180	104	126
Размер блока	18 Кбит	18 Кбит	18 Кбит
Общий объем	4824 КБит	1872КБит	2268кБит
Fmax	320МГц	250МГц	320МГц

В Spartan-6, также как и в Spartan-3, имеются два аппаратных умножителя 18x18бит и 48-битный аккумулятор, но их максимальная тактовая частота увеличена до 390 МГц. Spartan-6 содержат аппаратные контроллеры, позволяющие подключать внешнюю память DDR, DDR-2, DDR-3, LPDDR (Low power DDR). В результате теперь появилась возможность с помощью встроенного с САПР инструмента «MIG Generator» реализовать в ПЛИС SDRAM контроллер. Кроме того, все ПЛИС Spartan-6 LXT содержат аппаратное ядро PCI Express v.1.1, которое позволяет организовать соединение не только с компьютером, но и между двумя ПЛИС.

Одна из отличительных характеристик ПЛИС - возможность организации скоростного обмена данными. Микросхемы серии Spartan-6 LXT содержат от 2 до 8 высокоскоростных приемопередатчиков, каждый из которых позволяет организовать передачу данных по последовательному дифференциальному интерфейсу со скоростью до 3.125Гбит/с.

Существенным достоинством ПЛИС этого семейства является также возможность программной реализации процессора MicroBlaze. Соответственно, в данном семействе обновлены версии IP-ядер и модернизирован пакет проектирования XPS-EDK. В частности, Xilinx начала активно внедрять в свои проекты с MicroBlaze шину AXI с архитектурой ARM.

Основные технические характеристики семейств Spartan-6 приведены на рисунке 1.9 (см. также [1.5]).

Spartan-6 LX (1.2 В, 1.0 В)
Подсемейство, оптимизированное под применения в устройствах массового производства для реализации логики, ЦОС и интерфейсов памяти

		LX4	LX9	LX16	LX25	LX45	LX75	LX100	LX150
Логические ресурсы	Секции (4 6-LUT + 8 FF)	600	1 430	2 278	3 758	6 822	11 662	15 822	23 038
	Логические ячейки	3 840	9 152	14 579	24 051	43 661	74 637	101 261	147 443
	Триггеры	4 800	11 440	18 224	30 064	54 576	93 296	126 576	184 304
Память	Распределенная память (max, кбит)	75	90	136	229	401	692	976	1 355
	Блоки памяти BRAM (по 18 кбит)	12	32	32	52	116	172	268	268
	Общая емкость BRAM (кбит)	216	576	576	936	2 088	3 096	4 824	4 824
Синхронизация	Блоки синхронизации (CMT)	2	2	2	2	4	6	6	6
Ресурсы ввода-вывода	Контакты (max)	132	200	232	266	358	400	480	576
	Диф. Пары (max)	66	100	116	133	179	200	240	288
Встроенные аппаратные ядра	DSP48A1	8	16	32	38	58	132	180	180
	PCI Express® блок	-	-	-	-	-	-	-	-
	Контроллер памяти	0	2	2	2	2	4	4	4
	Трансивер GTP	-	-	-	-	-	-	-	-
Классы быстродействия	Коммерческий (C)	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3
	Индустриальный (I)	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3	-L1,-2,-3
Конфигурация	Конфигурационная память (Мбит)		2.7	3.7	6.4	11.9	19.6	26.5	33.8

Корпус	Размер/шаг	Максимальное число пользовательских контактов							
CPG196	8x8/0.5 мм	106	106	106					
TQG144	20x20/0.5 мм	102	102						
CSG225	13x13/0.8 мм	132	160	160					
CSG324	15x15/0.8 мм		200	232	226	218			
CSG484	19x19/0.8 мм					320	328	336	336
FT(G)256	17x17/1.0 мм		186	186	186				
FG(G)484	23x23/1.0 мм				266	316	280	326	336
FG(G)676	27x27/1.0 мм					358	408	480	498
FG(G)900	31x31/1.0 мм								576

Spartan-6 LXT (1.2 В)
Подсемейство, оптимизированное под применения в устройствах массового производства для реализации логики, ЦОС, интерфейсов памяти и высокоскоростных последовательных интерфейсов

		LX25T	LX45T	LX75T	LX100T	LX150T
Логические ресурсы	Секции (4 6-LUT + 8 FF)	3 758	6 822	11 662	15 822	23 038
	Логические ячейки	24 051	43 661	74 637	101 261	147 443
	Триггеры	30 064	54 576	93 296	126 576	184 304
Память	Распределенная память (max, кбит)	229	401	692	976	1 355
	Блоки памяти BRAM (по 18 кбит)	52	116	172	268	268
	Общая емкость BRAM (кбит)	936	2 088	3 096	4 824	4 824
Синхронизация	Блоки синхронизации (CMT)	2	4	6	6	6
Ресурсы ввода-вывода	Контакты (max)	250	296	348	498	540
	Диф. Пары (max)	125	148	174	249	270
Встроенные аппаратные ядра	DSP48A1	38	58	132	180	180
	PCI Express® блок	1	1	1	1	1
	Контроллер памяти	2	2	4	4	4
	Трансивер GTP	2	4	8	8	8
Классы быстродействия	Коммерческий (C)	-2,-3,-4	-2,-3,-4	-2,-3,-4	-2,-3,-4	-2,-3,-4
	Индустриальный (I)	-2,-3	-2,-3	-2,-3	-2,-3	-2,-3
Конфигурация	Конфигурационная память (Мбит)	6.4	11.9	19.6	26.5	33.8

Корпус	Размер/шаг	Максимальное число пользовательских контактов / Трансиверы GTP				
CSG324	15x15/0.8 мм	190/2	190/4			
CSG484	19x19/0.8 мм		296/4	292/4	296/4	296/4
FG(G)484	23x23/1.0 мм	250/2	296/4	268/4	296/4	296/4
FG(G)676	27x27/1.0 мм			348/8	376/8	396/8
FG(G)900	31x31/1.0 мм				498/8	540/8

Рисунок 1.9 - Основные характеристики семейства Spartan-6.

Литература к разделу 1:

1.1. http://www.xilinx.com

1.2. Кузелин М.О, Кнышев Д.А., Зотов В.Ю. Современные семейства ПЛИС фирмы XILINX. Справочное пособие. М.: Горячая линия - Телеком, 2004. – 440 с.

1.3. Тарасов И.Е. Разработка цифровых устройств на основе ПЛИС XILINX® с применением языка VHDL. М.: Горячая линия - Телеком, 2005. – 252 с.

1.4. Певцов Е.Ф., Смирнов Н.А. Проектирование цифровых схем на основе ПЛИС // Методические указания по выполнению лабораторных работ. Московский государственный институт радиотехники, электроники и автоматики (технический университет). М.: МИРЭА 2006, 32 с.

1.5. http://www.plis.ru

2. Порядок работы с программным пакетом ISE для автоматизированного проектирования систем на основе ПЛИС

Интегрированная система автоматизированного проектирования **Integrated Software Environment** (**САПР ISE**) является основным продуктом сквозного проектирования цифровых систем на базе ПЛИС фирмы Xilinx. САПР, поддерживает все выпускаемые ПЛИС и имеет несколько вариантов поставки. Вариант **ISE WebPACK** является свободно распространяемым, но имеет ограничения по максимальному логическому объему программируемых микросхем. Этот объем установлен на уровне 1,5 млн. эквивалентных логических вентилей, что включает в себя все недорогие ПЛИС. Для получения программного обеспечения WebPACK ISE следует обратиться в соответствующий раздел сайта производителя [2.1].

В данном разделе излагаются основные ведения о порядке работы с версией пакета **ISE Webpack** 13.2 (2011г.). Цель настоящего пособия – на основе примера проекта простейшего логического устройства продемонстрировать основные процедуры проектирования, чтобы максимально сократить промежуток времени между первым знакомством со средой проектирования ISE и конфигурированием проекта своего оригинального устройства в ПЛИС. Более подробно особенности работы и методика проектирования в среде ISE изложены в работах [2.2 … 2.5], а также в многочисленных публикациях в Интернете.

2.1 Пример проекта цифрового устройства в среде ISE (лабораторная работа №1)

2.1.1 Исходные данные

Проектом в САПР ISE является совокупность файлов, которые содержат информацию, необходимую и достаточную для выполнения всех этапов разработки цифрового устройства.

При разработке цифровых устройств на базе ПЛИС Xilinx условно можно выделить следующие основные этапы проектирования [2.2]:

- анализ задачи, разработка алгоритма работы устройства, разбиение проекта на модули, определение семейства ПЛИС, типа кристалла, корпуса, а также средств синтеза;
- разработка описания проектируемого устройства и его отдельных модулей в форме принципиальной схемы, кода поведенческого описания на языке HLD (Hardware Language Description);
- синтез модулей и всего устройства;
- функциональное моделирование;
- размещение и трассировка проекта в кристалле;
- оптимизация устройства по временным характеристикам, потребляемой мощности и ресурсам ПЛИС;
- загрузка проекта в кристалл (программирование ПЛИС);
- подготовка технической документации проекта.

Исходные данные для проектирования (техническое задание или спецификация устройства):

1. На основе микросхемы FPGA 6SLX45CSG324 семейства Spartan-6 разработать логическое устройство, реализующее функции 2НЕ-ИЛИ и 2И. При разработке устрой-

ства применить комбинацию встроенного библиотечного модуля 2И и модуля 2НЕ-ИЛИ на основе логического поведенческого описания.

2. В качестве макета для демонстрации разработки использовать плату ATLYS (см. [2.6]), содержащую большой комплект вспомогательных периферийных устройств для реализации проектов на основе микросхемы FPGA 6SLX45CSG324 семейства Spartan-6.

3. Входные сигналы формировать при помощи ползунковых выключателей платы ATLYS, подключенных к входам ПЛИС.

4. Для тестирования разработанного устройства его выходы подключить к светодиодам, расположенным на демонстрационной плате ATLYS.

2.1.2 Задание параметров нового проекта

Для создания проекта нового устройства следует запустить на исполнение основную программу среды **ISE Project Navigator** (например, из главного системного меню, как показано на рисунке 2.1).

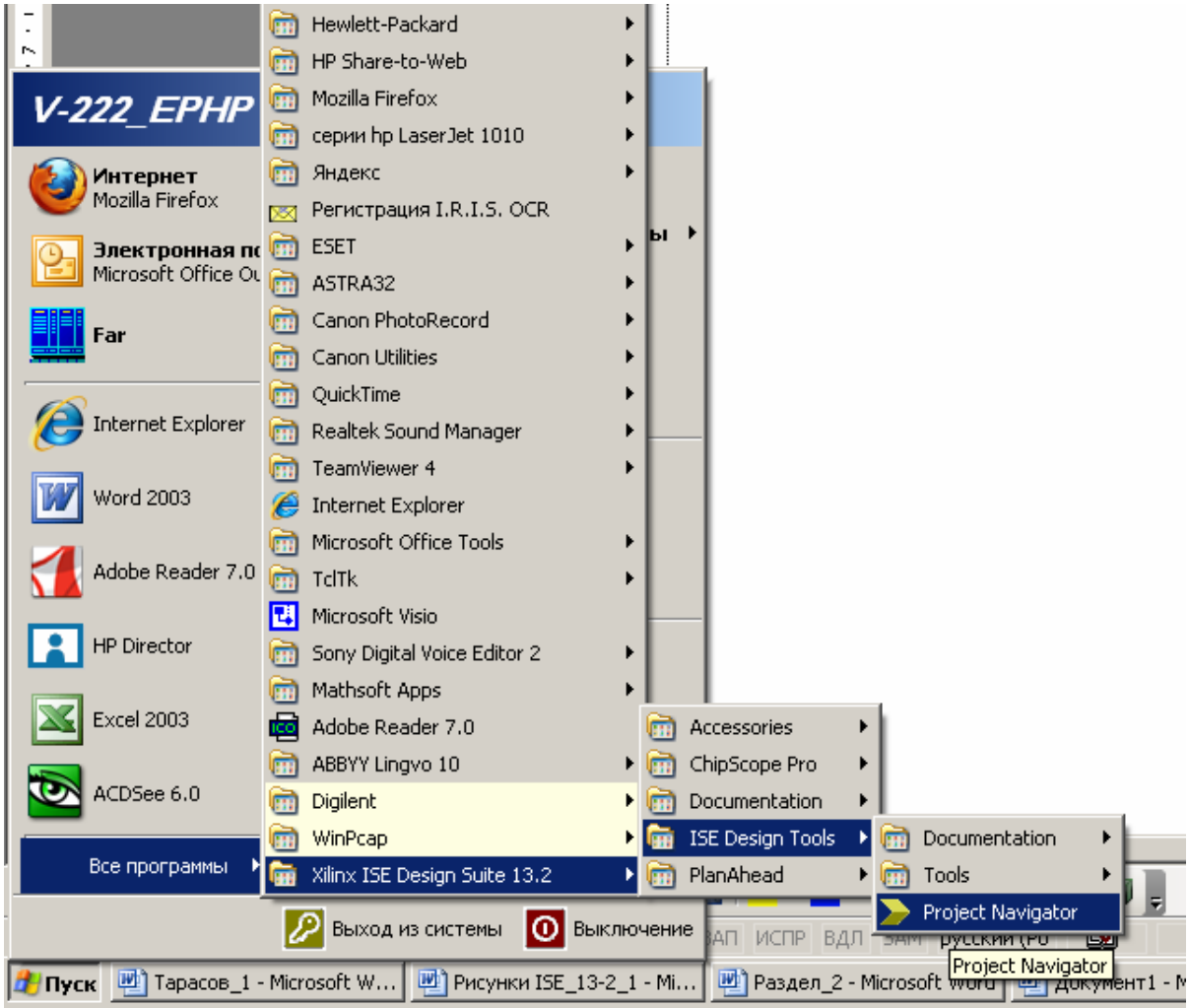


Рисунок 2.1 – Запуск программы **ISE Project Navigator** из главного системного меню.

В результате будет открыто основное окно (**Навигатор проектов ISE**), содер-

жащее данные предыдущего проекта (на рисунке 2.2. это проект **qwe**). Как показано на рисунке 2.2, при стандартных настройках интерфейс пользователя представляет собой комбинацию типичных для интегрированных сред проектирования окон:

– окно исходных модулей, т.е. непосредственно навигатор проекта;окно документов; окно процессов (проектных процедур), которые могут быть выполнены для модуля, выбранного в окне навигатора проекта;окно консоли сообщений о ходе выполнения проектных процедур и их результатах.

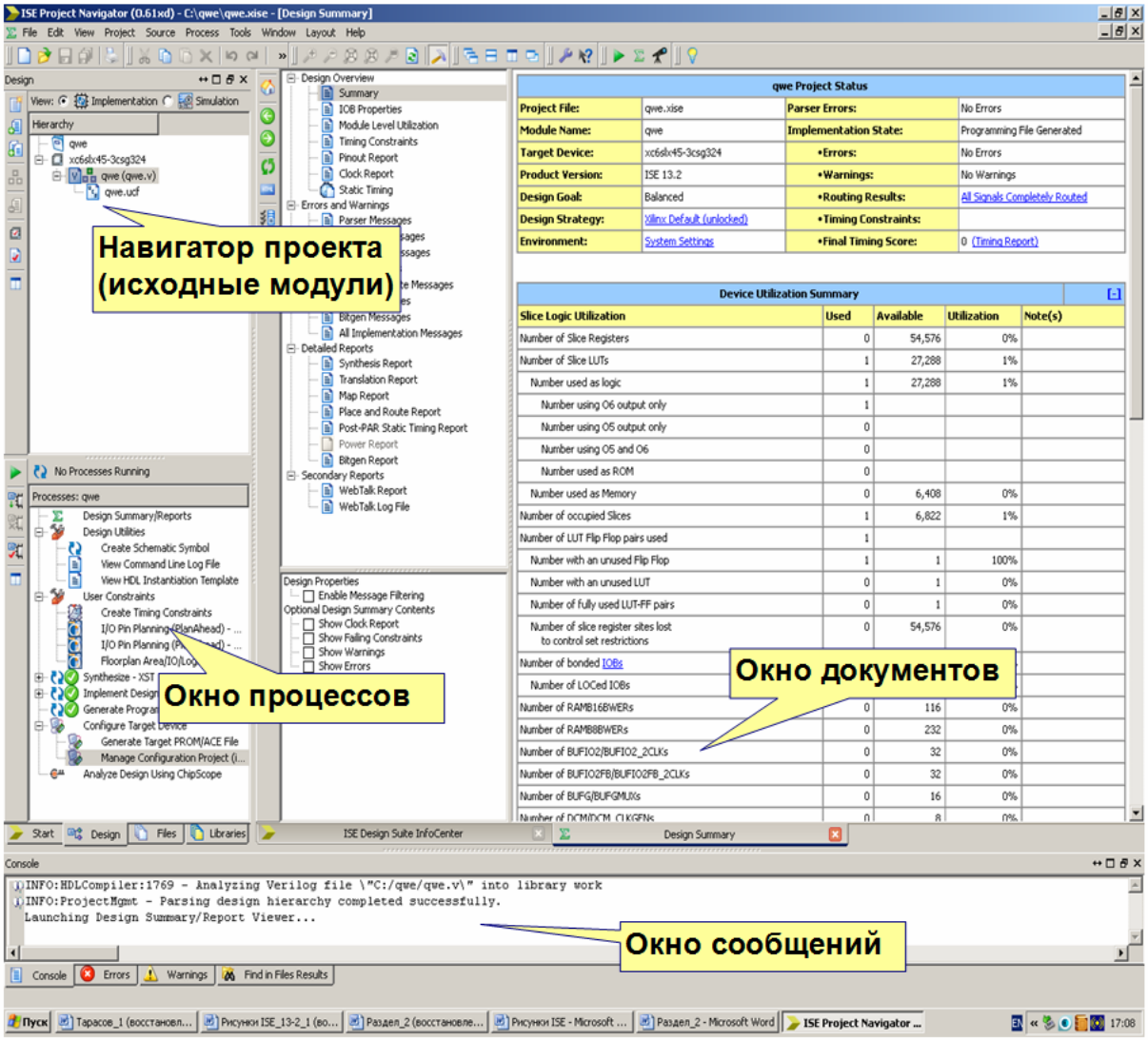


Рисунок 2.2 – Главное окно интерфейса пользователя **ISE Project Navigator.**

В свою очередь, окна могут иметь вкладки, раскрывающиеся при нажатии соответствующих кнопок, расположенных внизу окна. Сверху расположены главное меню и панель инструментов проекта, обозначенных пиктограммами и дублирующими вызов некоторых основных функций вкладок главного меню. Каждый пункт главного меню открывает всплывающий список соответствующих функций, перечень основных из них приведен в таблице 2.1.

Таблица 2.1 – Перечень основных функций главного меню **ISE Project Navigator.**

Пункт меню	Соответствующие инструменты подменю:	
File	***Работа с файлами, функции печати и завершения работы**:*	
	Создать новый проект	***New Project …***
	Открыть уже существующий проект	***Open Project…***
	Открыть один из примеров проектов	***Open Example…***
	Просмотреть содержимое проекта	***Project Browser…***
	Скопировать проект	***Copy Project …***
	Закрыть проект	***Close Project***
	Сохранить проект	***Save***
	Распечатать проект	***Print***
	Открыть последний проект	***Recent Project***
	Выход	***Exit***
Edit	***Работа с встроенным редактором HDL кода и функции настройки конфигурации (Preference) навигатора проекта:***	
	Отменить действие	***Undo***
	Вернуть отмененное действие	***Redo***
	Вырезать	***Cut***
	Копировать	***Copy***
	Вставить	***Paste***
	Удалить	***Delete***
	Найти…	***Find…***
	Найти в данном файле	***Find in file***
	Ввести комментарий	***Comment***
	Преобразовать	***Convert***
	Вставить файл	***Insert file***
	Перейти к…	***Go to***
	Использовать шаблон кода языка (TCL, UCF, VHDL, Verilog)	***Language template… ((Tcl, UCF, VHDL, Verilog)***
	Выбрать все	***Select all***
	Пользовательские настройки отображения кода…	***Preference…***
View	***Функции настройки режимов просмотра***	
	Отображать на экране окна с результатами работы	***Panel =>***
	Отображать на экране меню инструментов	***Toolbar =>***
	Отображать строку состояния	***Status bar***
	Отображать описание в окне консоли	***Transcript***

Продолжение Таблицы 2.1.

	Увеличить или уменьшить масштаб отображения в активном окне	***Zoom***
	Закладки	***Bookmark***
	Режим отображения служебных символов	***Show…***
	Отображать строки кода	***Line Number***
	Обновить окно	***Refresh***
Project	***Функции работы с проектом***	
	Создать новый модуль	***New source…***
	Добавить модуль…	***Add source…***
	Задать порядок компиляции файлов проекта вручную	***Manual compile order***
	Загрузить файл настройки порядка компиляции проекта	***Import Custom Compile File List…***
	Удалить файлы проекта, созданные ISE	***Cleanup Project Files…***
	Создать архив всех файлов проекта (.zip)*	***Archive…***
	Создать скрипт (файл с последовательностью команд для выполнения проектных процедур) на языке TCL	***Generate Tcl Script…***
	Настроить цели и стратегию проекта	***Design Goals & Strategies…***
	Документ отчета о проекте и отчеты о выполнении проектных процедур	***Design Summary/Reports***
	Настройки проекта…	***Design Properties…***
Source	***Создание файлов, входящих в состав проекта и работа с ними, установка параметров проекта***	
	Открыть	***Open***
	Задать как модуль верхнего уровня иерархии	***Set as Top Module***
	Использовать технологию SmartGuide (интеллектуальный проводник) для повторного синтеза проекта	***SmartGuide***
	Удалить из проекта, но сохранить в папке	***Remove***
	Переместить файл в другую библиотеку…	***Move to Library…***
	Преобразовать из TBL в HDL тестовый стенд	***Convert TBW to HDL testbench…***
	Настройка свойств исходных файлов проекта…	***Sources Properties…***
Process	***Управление процедурами проектирования***	
	Запустить на исполнение процесс синтеза файла с описанием верхнего уровня проекта…	***Implement Top Module…***
	Запустить процесс на исполнение	***Run***
	Остановить выполнение процесса	***Stop***
	Запустить на исполнении процесс с имеющимися данными	***Run With Current Data***

Продолжение Таблицы 2.1.

	Настройка параметров выполнения процесса	***Process Properties…***
	Изменить статус процесса на основе перезагрузки данных…	***Force Process Up-to-Date***
Tools	***Вызов инструментов при выполнении процедур проектирования***	
	Редактирование проектных ограничений…	***Constraints Editor…***
	Генератор ядра…	***Core Generator…***
	Открыть программу PlanAhead	***PlanAhead =>***
	Просмотреть схему…	***Schematic Viewer =>***
	Открыть программу анализа временных параметров	***Timing Analyzer =>***
	Открыть программу редактирования параметров FPGA	***FPGA Editor =>***
	Открыть программу анализа потребляемой мощности	***XPower Analyzer…***
	Открыть программу конфигурирования ПЛИС	***iMPACT…***
	Открыть программу управления параллельными потоками синтеза	***SmartXplorer =>***
Window	***Работа с окнами навигатора проекта***	
	Новое окно	***New Windows***
	Закрыть окно	***Close***
	Следующее окно	***Next***
	Предыдущее окно	***Prevision***
	Расположить каскадом	***Cascade***
	Сделать окно плавающим	***Float***
	Список окон	***Windows List***
Layout	***Управление расположением окон навигатора проекта***	
	Восстановить конфигурирование по умолчанию	***Load Default Layout***
	Сохранить данную конфигурацию как …	***Save Layout As…***
	Экспортировать данную конфигурацию…	***Export Layout…***
	Импортировать данную конфигурацию…	***Import Layout…***
Help	***Вызов справочной системы***	
	Разделы справки	***Help Topic***
	Учебник по САПР	***Software Manual***
	Найти в учебнике	***Search Software Manual…***
	Справка о Xilinx в Интернете	***Xilinx on the Web =>***
	Справочный центр ISE	***ISE Design Suite Info-Center***
	Особенности новых версий…	***Key New Features …***
	Помощь в диалоговом режиме через сеть	***WebTalk Help***
	Менеджер лицензий…	***Manger License…***

После запуска Навигатора проекта в нем следует создать новый проект, выбрав

на главном меню функцию **Project** и в открывшемся списке **New Project.** В соответствующем диалоговом окне, представляющем собой форму базы данных проекта (см. рисунок 2.3), следует указать имя проекта, которое будет совпадать с именем каталога, создаваемого для хранения всех требуемых для работы файлов.

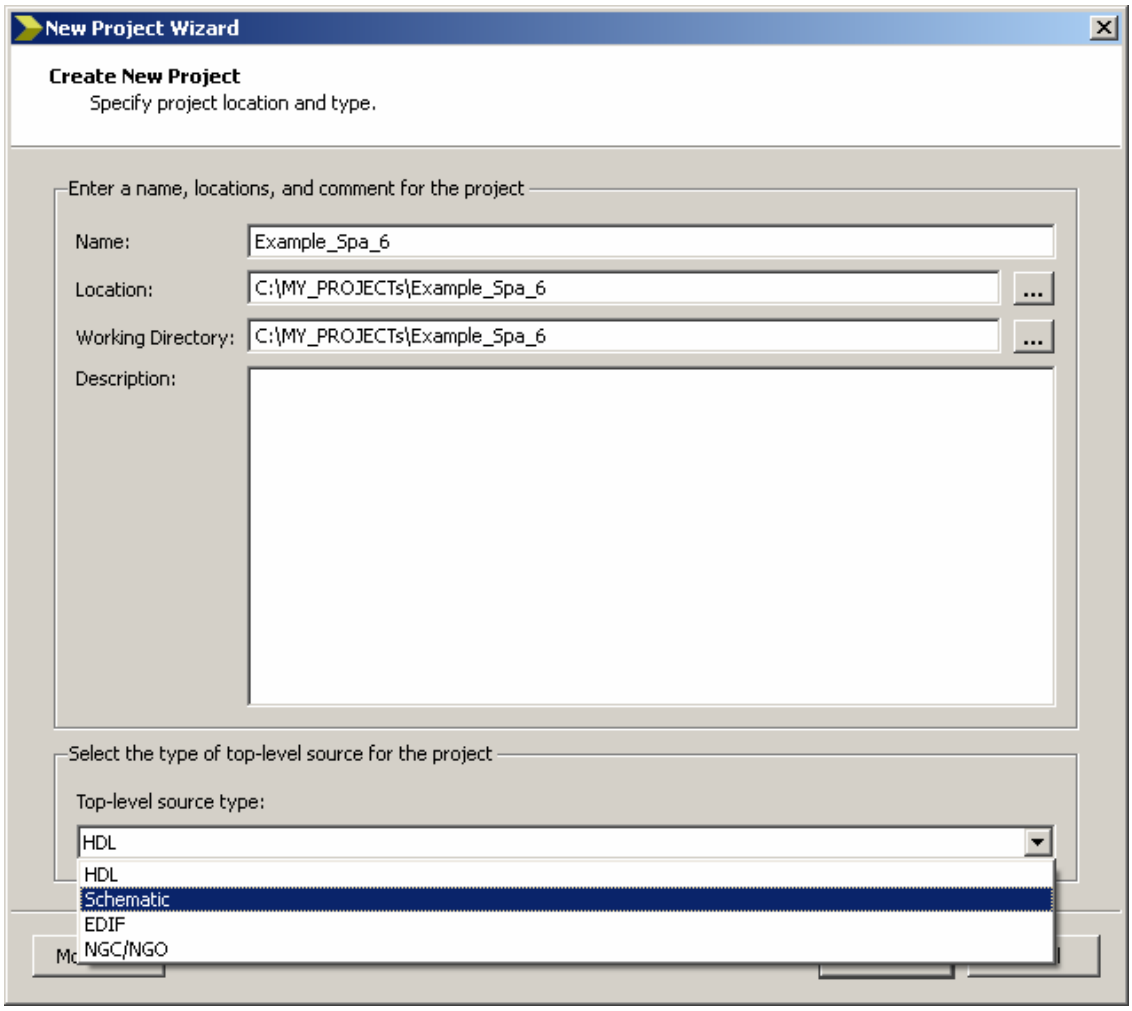


Рисунок 2.3 – Диалоговое окно создания нового проекта.

Особенности САПР требуют, чтобы в пути к рабочим файлам не было пробелов и символов кириллицы, поэтому следует сразу же проследить, чтобы проект **не был** размещен в каталоге вида ***C:\Мои документы***. Кроме того, следует проследить, чтобы проект располагался **вне** папки, в которой размещен сам пакет САПР Xilinx ISE. На рисунке показано, что для проектов САПР ISE выделен каталог ***C:\MY_PROJECTs\Example_Spa_6***.

В этом же окне следует определить параметр ***Top-level source type*** (тип файла верхнего уровня иерархии), который задает формат представления «главного модуля проекта», в который будут вложены другие модули проекта. Выводы этого модуля будут подключены к выводам ПЛИС. В списке показаны четыре типа такого файла:

- ***HDL*** (*Hardware Description Language*) означает, что файлом верхнего уровня является текстовый файл на языке описания аппаратуры;

- ***Schematic*** – файл верхнего уровня представляет собой графическое изображение

принципиальной электрической схемы, составленной из стандартных библиотечных модулей, и модулей, добавляемых разработчиком в виде других графических схем или файлов на HDL;

- ***EDIF, NGC/NGO*** – устройство представляется в виде готовых списков связей, разработанных ранее в САПР ISE или с помощью иных программных инструментов. Маршруты, основанные на EDIF и NGC/NGO, представляют интерес в том случае, если в ПЛИС выполняется устройство, приобретенное в виде IP-ядра. В такой проект невозможно внести несанкционированные изменения, или восстановить его схему, имея NGC-представление.

На начальном этапе освоения САПР ISE для освоения маршрута проектирования следует выбирать схемотехническое представление верхнего уровня, поскольку оно наглядно представляет структурную схему проекта.

В следующем диалоговом окне (рисунок 2.4) следует указать наименование микросхемы ПЛИС, которая будет использована для выполнения проекта.

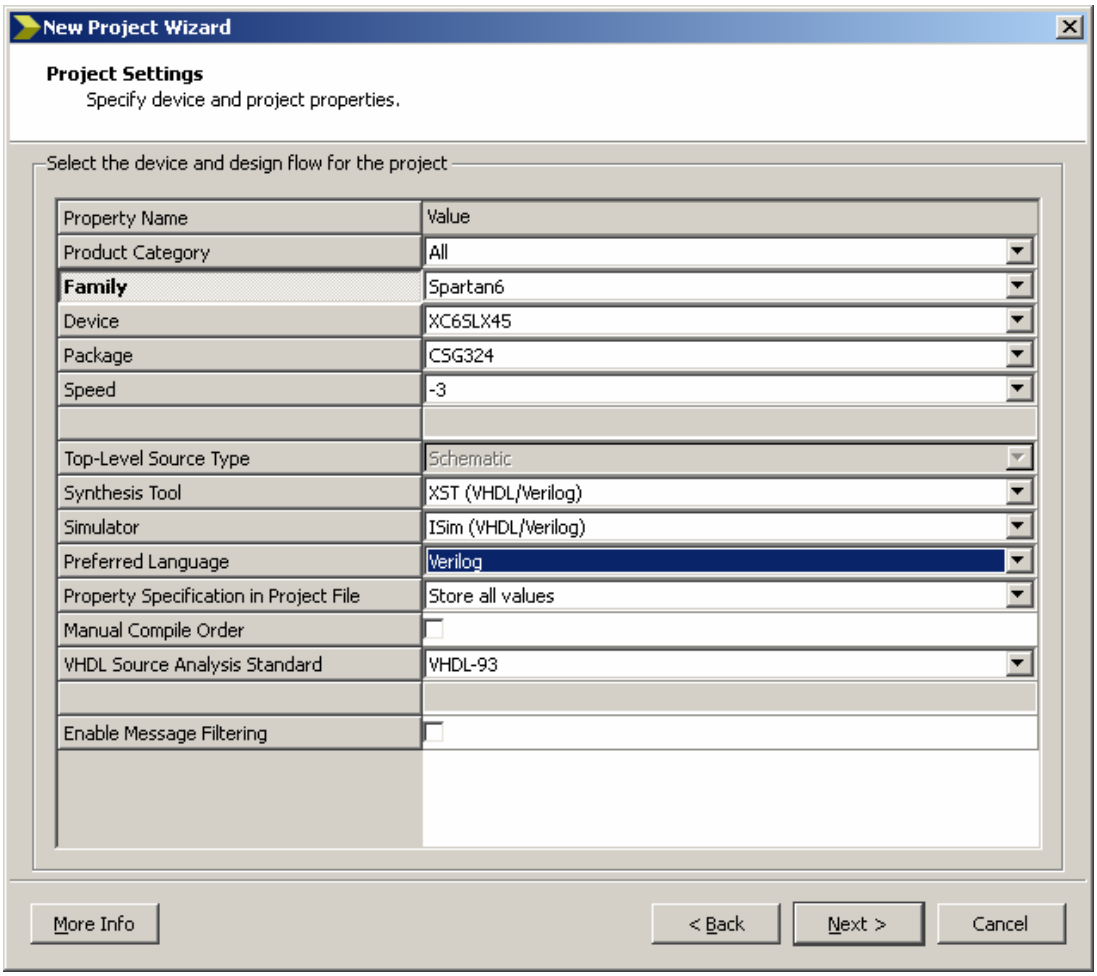


Рисунок 2.4 – Настройка параметров ПЛИС при создании нового проекта САПР ISE.

Ее тип впоследствии можно будет изменить. В соответствии с требованиями технического задания для выполнения данного проекта используется демонстрационная плата ATLYS, предназначенная для освоения проектирования цифровых систем в ПЛИС Xilinx. На плате установлена микросхема Spartan-6 LX45 в корпусе CSG324 с классом скорости (*speed grade*) -3. Здесь же следует выбрать из предложенных списков

тип инструментов синтеза XST(VHDL/Verilog), программу для моделирования iSim(VHDL/Verilog), язык описания аппаратуры (Verilog). Итоговый вид правильно заполненных полей окна настройки параметров ПЛИС показан на рисунке 2.4.

Следующее окно (рисунок 2.5) содержит форму отчета, обобщающую исходные данные проекта. Соответствующий текстовый файл будет сохранен автоматически в каталоге проекта для последующего использования в проекте. В этой форме пользователю предлагается проверить исходные данные. При необходимости корректировки, нажав кнопку **Back**, можно вернуться к предыдущим формам и исправит их.

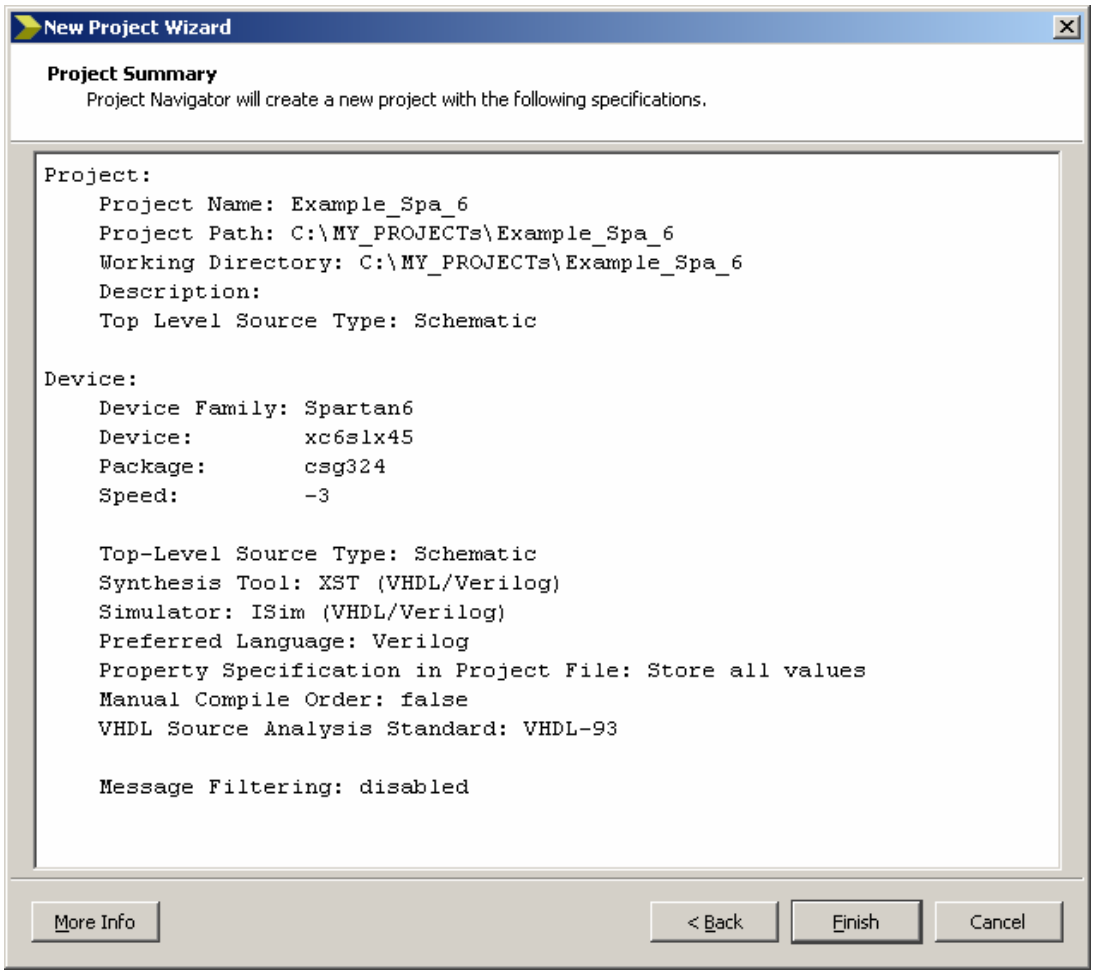


Рисунок 2.5 – Отчет об исходных данных проекта.

После нажатия кнопки **Finish** в окне исходных модулей навигатора проекта появится название текущего проекта и тип выбранной микросхемы.

2.1.3 Принципиальная схема устройства и ее компоненты

Для продолжения следует вызвать функцию **Project – New Source** и в диалоговом окне создания новых модулей проекта назначить форму представления файла верхнего уровня проекта (**Schematic**) и дать ему название (в данном случае - ***Top_Ex_Spa6***) как это показано на рисунке 2.6.

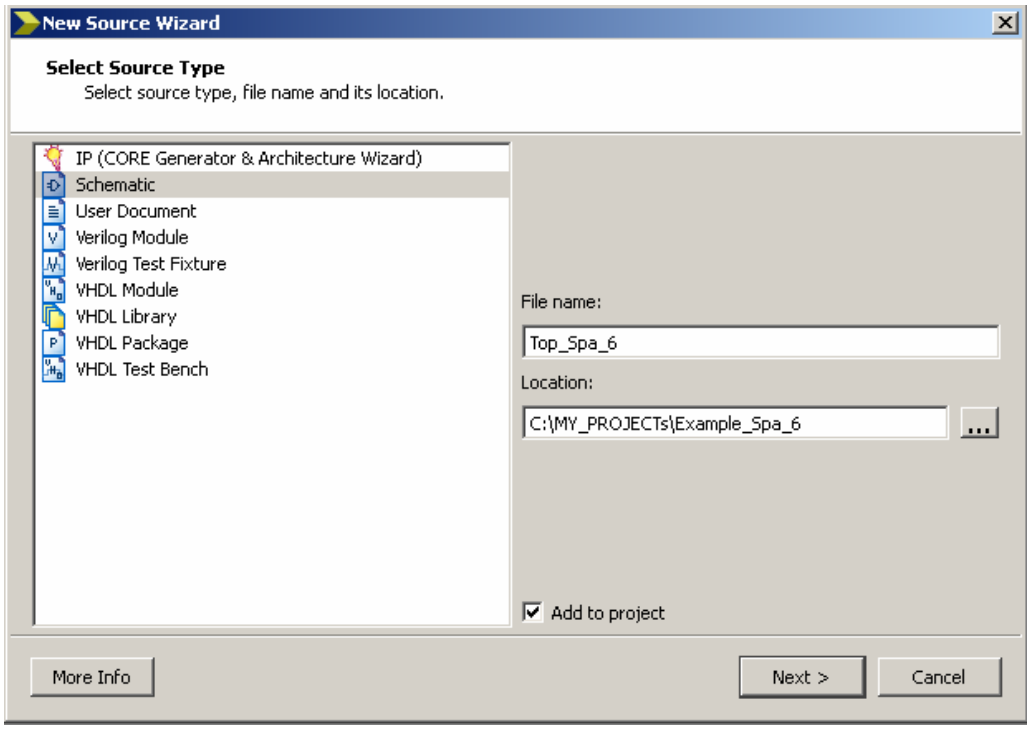


Рисунок 2.6 – Диалоговое окно мастера создания нового компонента (модуля) проекта.

Далее следует перейти к следующей форме (кнопка **Next**), проверить, что текстовый файл с описанием модуля верхнего уровня содержит правильные данные (см. рисунок 2.7), после чего перейти к следующей процедуре проектирования, нажав кнопку **Finish**.

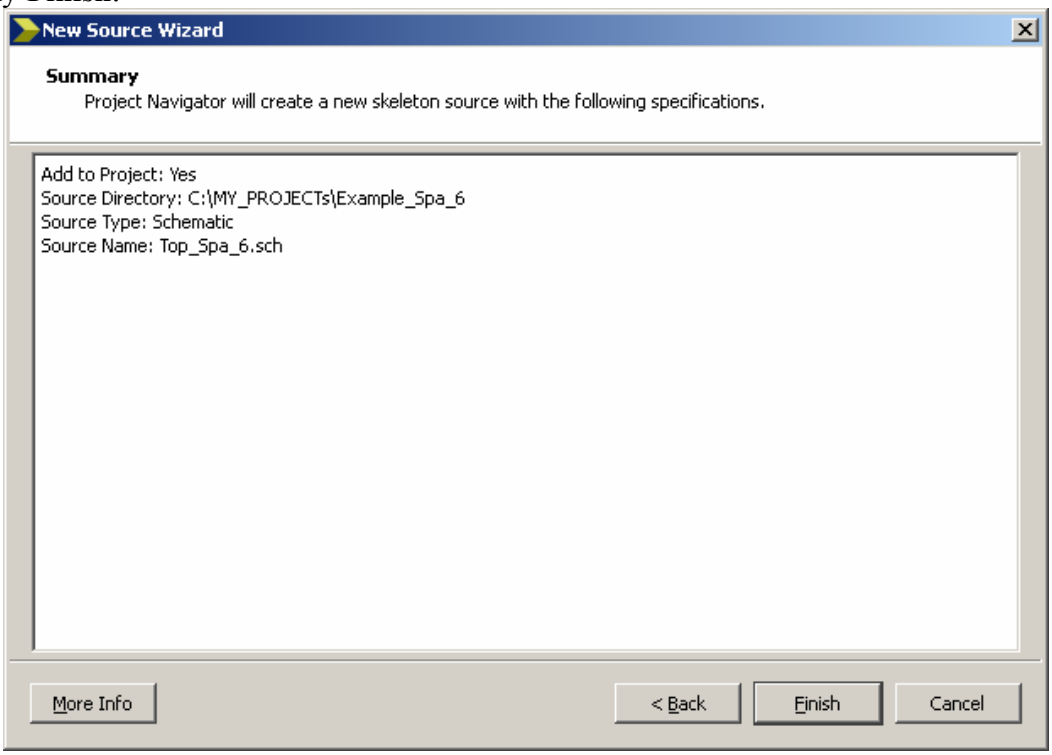


Рисунок 2.7 – Отчет о создании нового компонента.

Вид навигатора проекта после создания файла верхнего уровня представлен на рисунке 2.8.

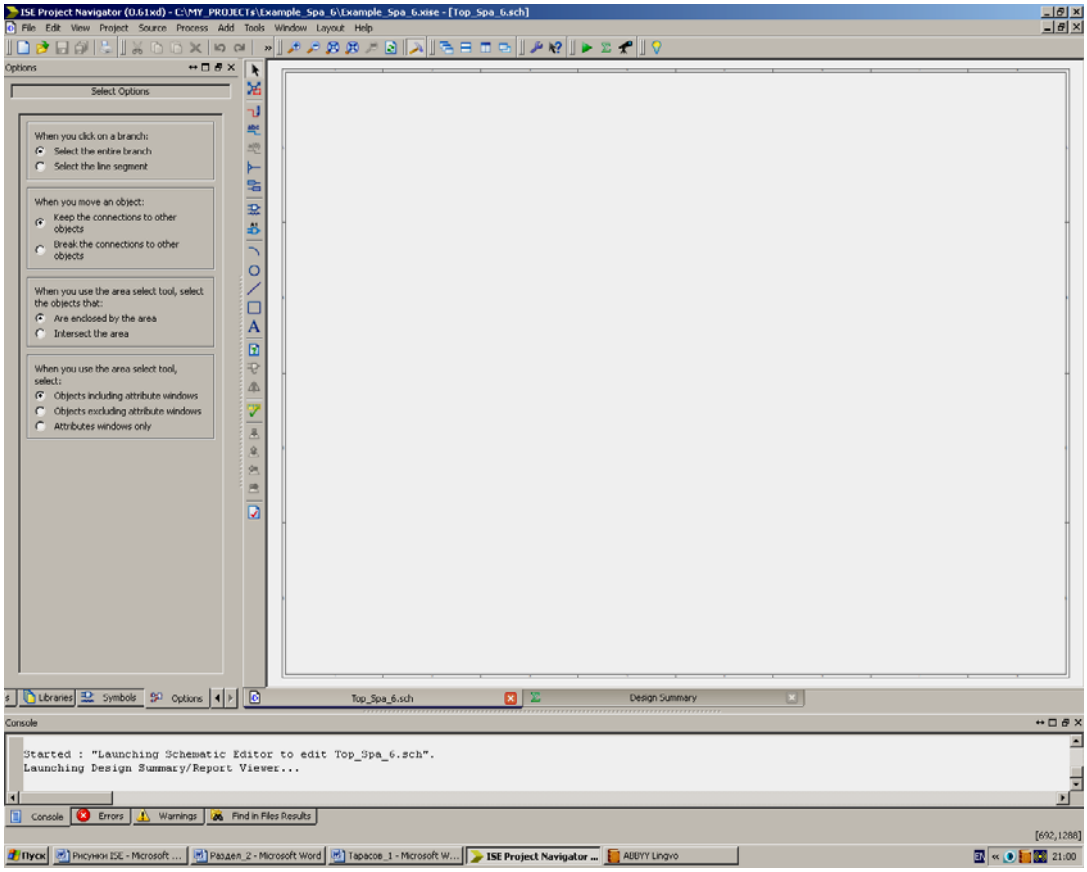


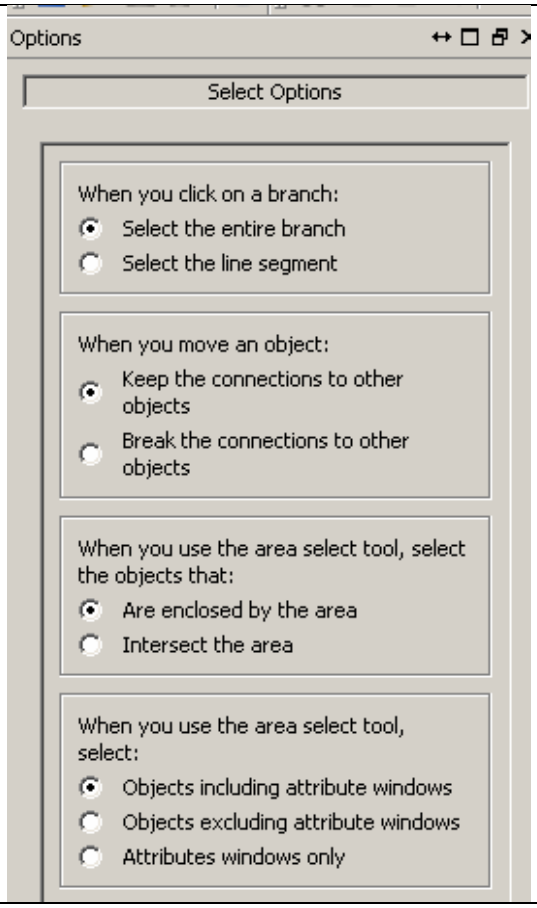
Рисунок 2.8 – Окно навигатора проекта с открытым документом графического редактора.

Поскольку для верхнего уровня проекта выбран тип модуля в виде принципиальной схемы (расширение файла модуля - ***.sch**), то соответствующее ему окно документа представляет собой графический редактор принципиальных схем ECS (Engineering Capture Schematic). Слева открыта вкладка **Option**, предназначенная для настроек режимов выбора объектов графического редактора с помощью курсора. Назначение переключателей этой вкладки разъясняет таблица 2.2.

Необходимо обратить внимание, что на месте расположения списка файлов расположено окно с несколькими вкладками (**Design, Symbols, Option, Library** и др). Собственно список файлов вызывается при открытии вкладки **Design**, а вкладка **Symbols** содержит список графических компонентов, разбитых на группы, как можно видеть на рисунке 2.9. Полная информация о стандартных ячейках заранее занесена в базу данных ISE**.**

Панель инструментов графического редактора ECS содержит стандартные для Windows-приложений кнопки-пиктограммы работы с файлами и кнопки специальных операций для редактирования принципиальных схем. Назначение этих кнопок поясняется таблицей 2.3.

Таблица 2.2 – Параметры настройки выбора объектов в графическом редакторе ECS.

	При выборе проводника: • выделять весь проводник; о выделять сегмент линии. При перемещении объекта: • сохранять соединения с другими объектами; о разрывать соединения с другими объектами. При выборе области, выделять объекты: • полностью расположенные в области; о пересекающие область. При выборе области, выделять: • объекты, включая окна атрибутов; о объекты, исключая окна атрибутов; о только окна атрибутов.

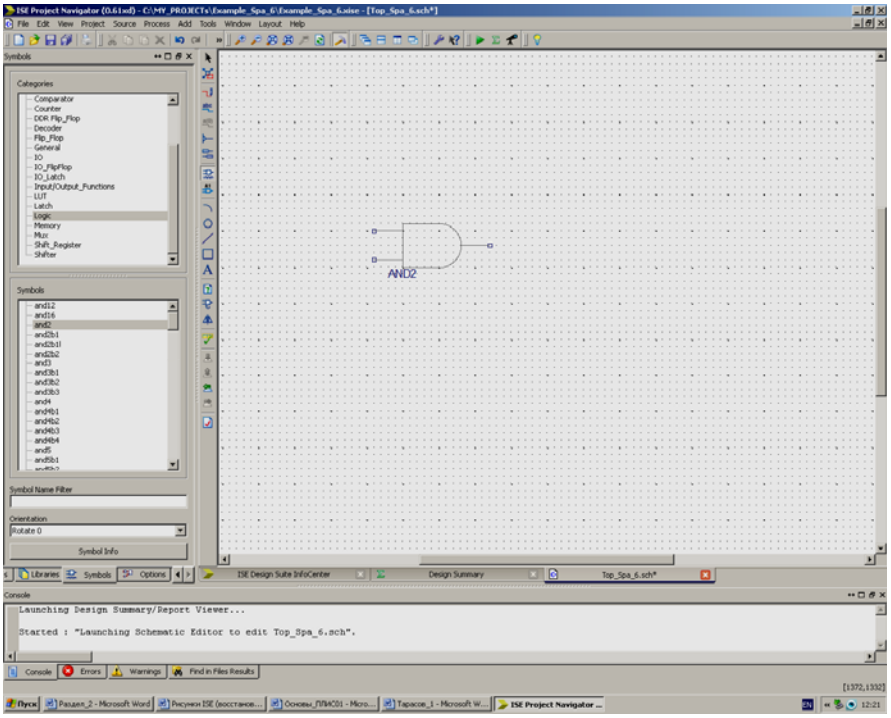


Рисунок 2.9 – Черновик принципиальной схемы модуля проекта с библиотечным элементом 2И.

Таблица 2.3 – Основные кнопки панели инструментов редактора схем.

Пиктограммы	Назначение элементов управления
	размещение дополнительных окон ECS
	вызов библиотеки шаблонов
  	управление режимами курсора и рисования схемы – выбор объектов, рисование проводников или шин, ввод имени одиночной линии, имени шины, подключение одиночных выводов от шины (bus taps), назначение маркеров ввода/вывода (I/O markers), выбор и размещение компонента (его также можно выполнить, открыв закладку **Symbols**, расположенную внизу окна менеджера проектов), ввод надписи
	элементы оформления схемы
	вызов справочной системы
	поворот и зеркальное отображение выбранного элемента
	проверка правильности изображения схемы
	переходы к уровням иерархии проекта
	отображение текущего вида схемы – изменение масштаба, переход к общему виду всего листа, выбор фрагмента для отображения

Чтобы разместить на схеме графический символ компонента, следует щелкнуть левой кнопкой мыши на его наименовании, после чего его можно захватить и поместить на схему его графическое обозначение. На рисунке 2.9 на схему, как и требуется в техническом задании, добавлен компонент 2И ***(and2)***, выбранный из группы ***logic*** библиотеки стандартных ячеек.

Для выполнения следующего пункта задания следует создать и добавить в проект новый модуль 2ИЛИ-НЕ, выполненный на основе логического описания на языке Verilog. Для этого следует вернуться к вкладке Design, вновь вызвать из главного меню функцию **Project – New Source,** задать тип нового модуля (**Verilog Module**) и присвоить ему имя (на рисунке 2.10 – модуль **My_logic**).

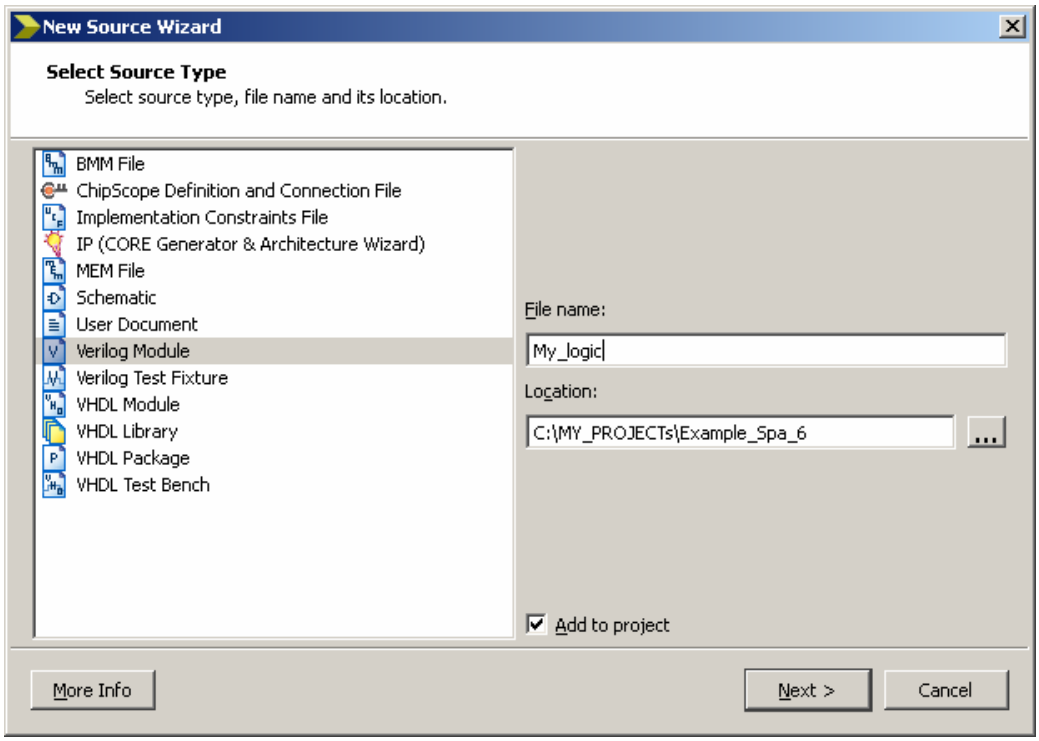


Рисунок 2.10 – Добавление в проект модуля **My_logic**, представляющего собой код поведенческого описания на языке Verilog.

На следующем шаге следует определить интерфейс создаваемого логического модуля, указав параметры его соединений с другими модулями, т.е. имена и типы его портов. Пример назначения параметров модуля приведен на рисунке 2.11.

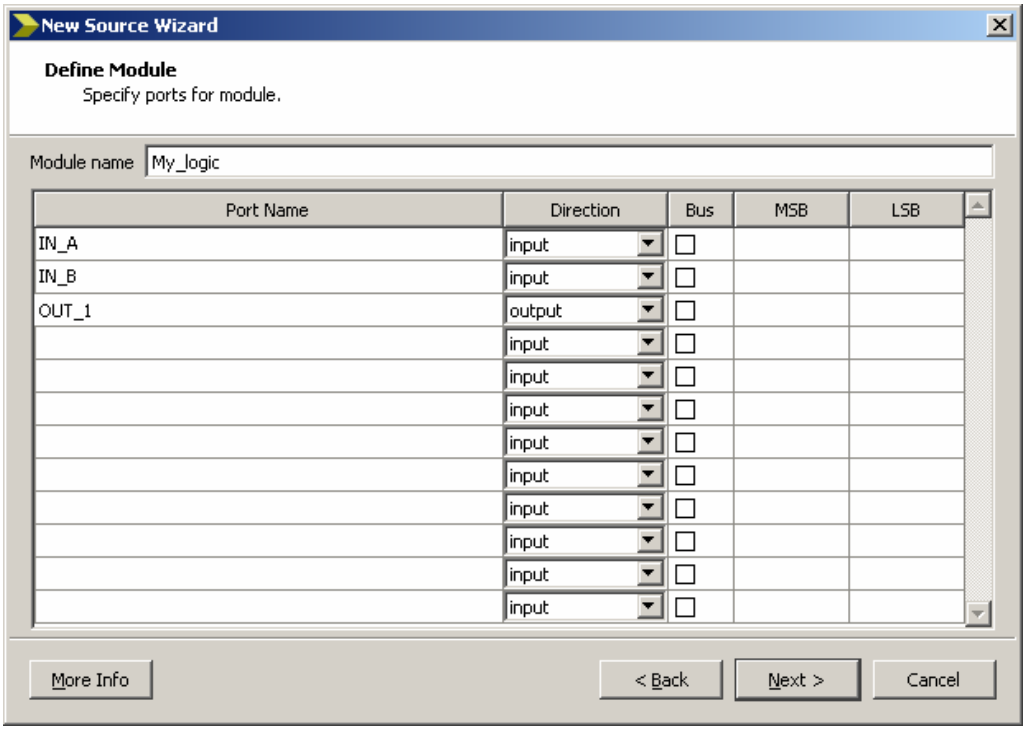


Рисунок 2.11 – Пример настройки параметров интерфейса модуля в САПР ISE.

Далее следует нажать кнопку **Next** и проверить правильность введенных данных, изучив соответствующий отчет. После нажатия кнопки **Finish**, в навигаторе проекта появится новый модуль, а в окне документов станет доступным редактирование шаблона кода его описания. Работа с редактором кода аналогична работе с любым текстовым редактором. При вводе стандартных функций и директив языка для удобства работы цвет текста автоматически изменяется. Как показано на рисунке 2.12, для выполнения задания (создать элемент, реализующий функцию 2ИЛИ_НЕ) следует вписать с шаблон строчку с кодом:

```
assign OUT_1 = ~(IN_A | IN_B);
```

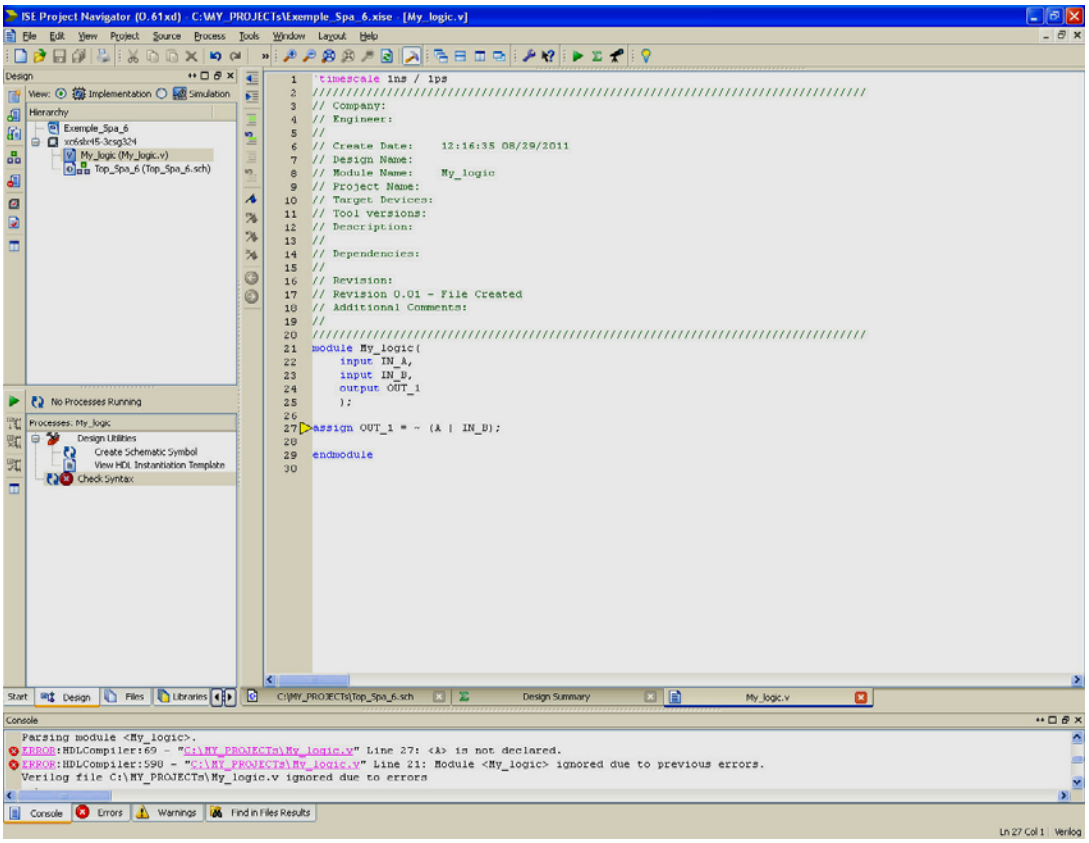


Рисунок 2.12 – Редактирование кода поведенческого описания модуля 2ИЛИ-НЕ. В консоли сообщений выведена информация об ошибке: попытка выполнить действия с необъявленной переменной «**A**».

Отредактированный код поведенческого описания следует сохранить (меню **File – Save**). Затем следует выполнить проектную процедуру проверки синтаксиса кода. Для этого следует выделить в окне модулей проекта соответствующий файл, и в окне **Process** и дважды щелкнуть на строчке вызова приложения **Check Syntax** (см. рисунок 2.12). Результаты выполнения проверки синтаксиса будут выведены в окно **Console** внизу навигатора проекта. При сообщении об ошибках (*Process "Check Syntax" failed*) следует найти в тексте консоли соответствующие строки (ошибки выделены цветом, как показано на рисунке 2.12) с указанием места, где в окне документов локализована ошибка, и типа ошибки. Ошибки необходимо исправить, после чего следует сохранить файл с исправленным кодом и снова проверить его синтаксис. Если проверка синтаксиса прошла успешно, то напротив строки с кнопкой вызова приложения **Check Syntax**

появится зеленый кружок с галочкой, а в консоли сообщений – строка *Process "Check Syntax" completed successfully*. После этого следует выполнить процедуру создания символьного представления разработанного логического модуля (в данном случае – модуля ***My_logic.v***).

Для создания символьного представления модуля и включения его в принципиальную схему следует выполнить следующие действия.

1) Выделить в окне исходных модулей проекта строку с названием модуля «***My_logic (My_logic.v)***», после чего в окне **Process** откроется в список процессов, доступных для данного модуля.

2) Дважды щелкнуть на строке **Design Utilites - Create Schematic Symbol**, проверить, что в окне консоли появилось сообщение об успешном выполнении компиляции графического символа логического модуля. Важно проверить корректность выполнения данной операции после каждого изменения кода проектируемого модуля и перезаписи соответствующего текстового файла кода, т.к. при ошибках САПР ISE будет всегда использовать в проекте последнюю корректно оттранслированную версию компонента. Впоследствии это послужит источником ошибок в работе всего проекта, который будет трудно обнаружить.

3) Открыть из окна модулей процесса документ с принципиальной схемой проекта, раскрыть закладку **Symbol** и убедиться, что в списке библиотечных элементов проекта в категории «·····<= **All symbols** =>» появился модуль **My_logic** (см. рисунок 2.13).

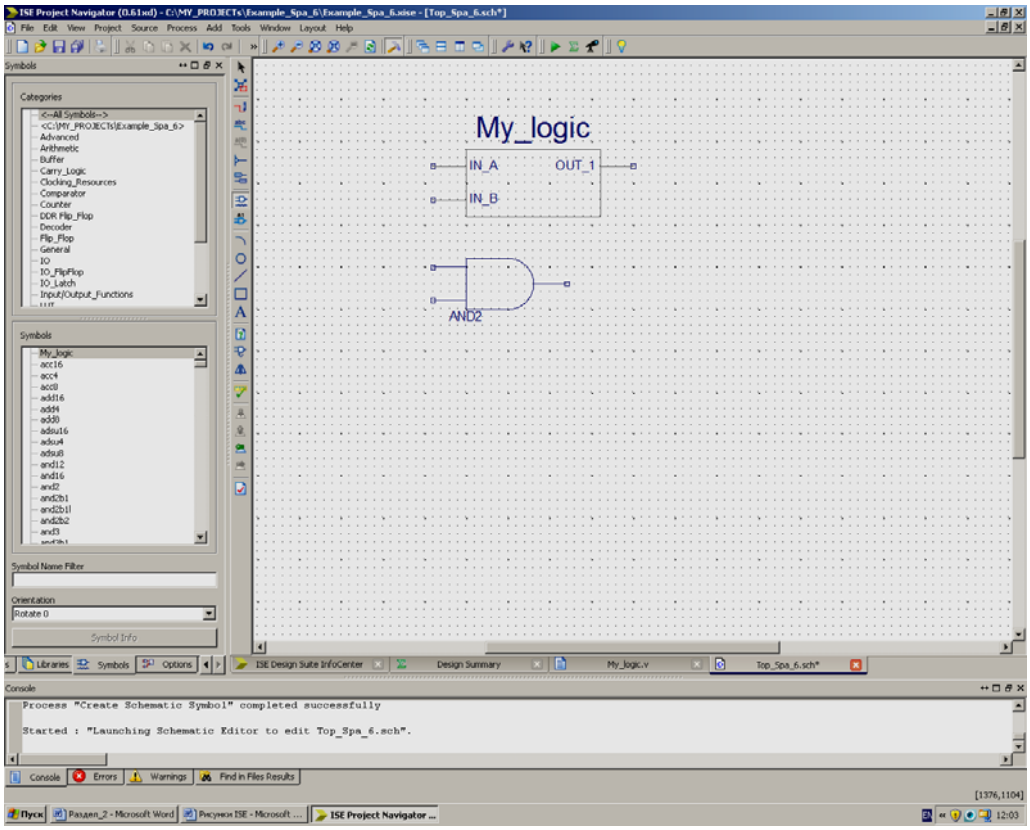


Рисунок 2.13 – Пример размещения на принципиальной схеме логического модуля **My_logic**.

Компоненты, созданные пользователем, сохраняются в отдельной группе, название которой соответствует пути к папке проекта. Поэтому, в частности, нужно, чтобы проект размещался вне общей папки с САПР ISE. Чтобы поместить символьное пред-

ставление модуля на принципиальную схему, как это показано на рисунке 2.13, нужно щелкнуть левой кнопкой мыши на его наименовании и захватив мышкой, перенести на схему. В принципиальную схему следует также внести необходимые стандартные элементы (см. пример на см. рисунке 2.14).

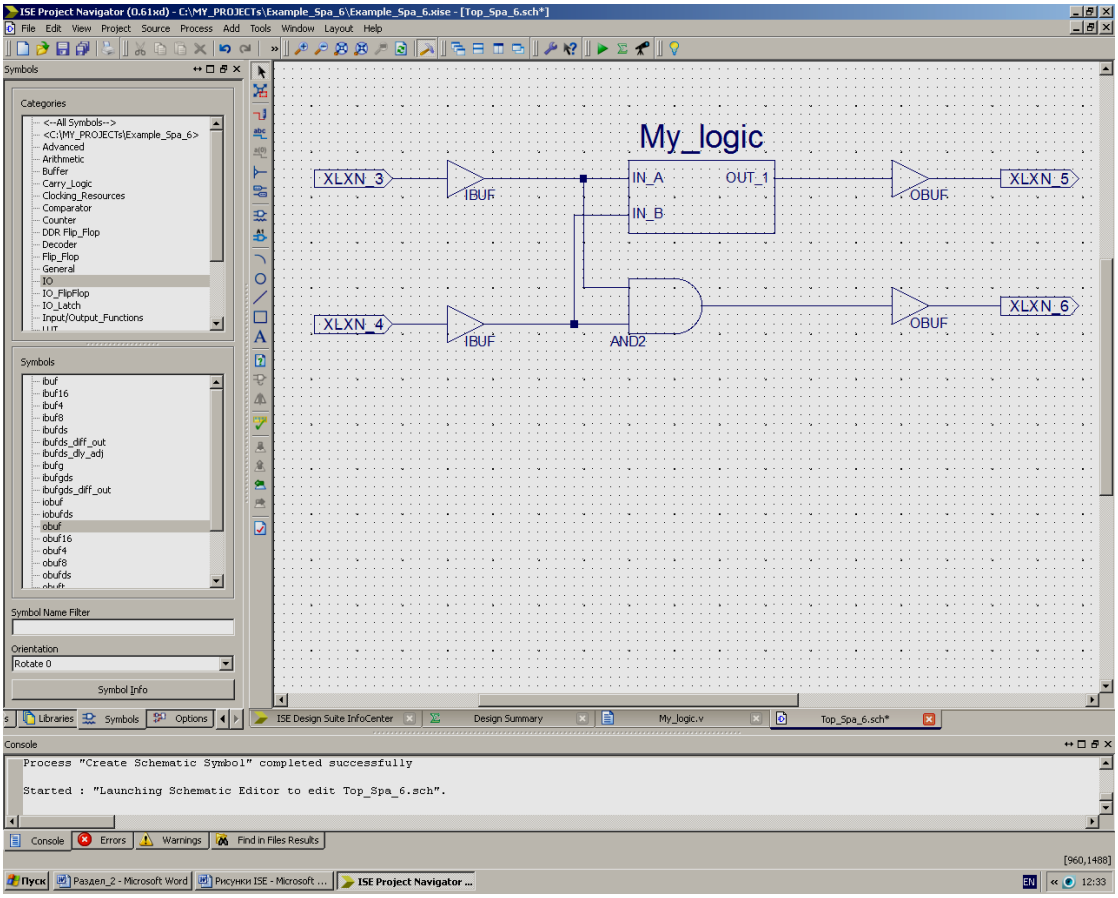


Рисунок 2.14 – Пример черновика принципиальной схемы разрабатываемого устройства. Требуется отредактировать названия входов.

В данном случае в соответствии с требованиями задания, нужно:4) Сначала входы модулей **My_logic** и **And2** соединить проводниками (использовать элементы ***Add Wire,*** левую кнопку мыши и клавишу «Esc»).

5) Выбрать из категории «·····**IO**» и перенести на принципиальную схему буферные стандартные элементы «***ibuf***» и «***obuf***» и соединить их проводниками с соответствующими узлами схемы.

6) Поместить на схему специальные маркеры входов и выходов, которые должны быть впоследствии подключены к выводам микросхемы, в которую будет «зашито» проектируемое устройство.

7) По умолчанию маркеру присваивается имя XLLN_XX, где XX – порядковый номер компонента в соответствии с внутренней нумерацией графического редактора ECS. Для окончательного оформления схемы рекомендуется дать маркерам входов и выходов собственные контекстные имена, отражающие назначение подключенным к ним сигналов. Для этого можно использовать контекстное меню маркеров, изменив имя порта (см. рисунок 2.15) или свойства выделенного объекта схемы, изменив имя узла (см. рисунок 2.16).

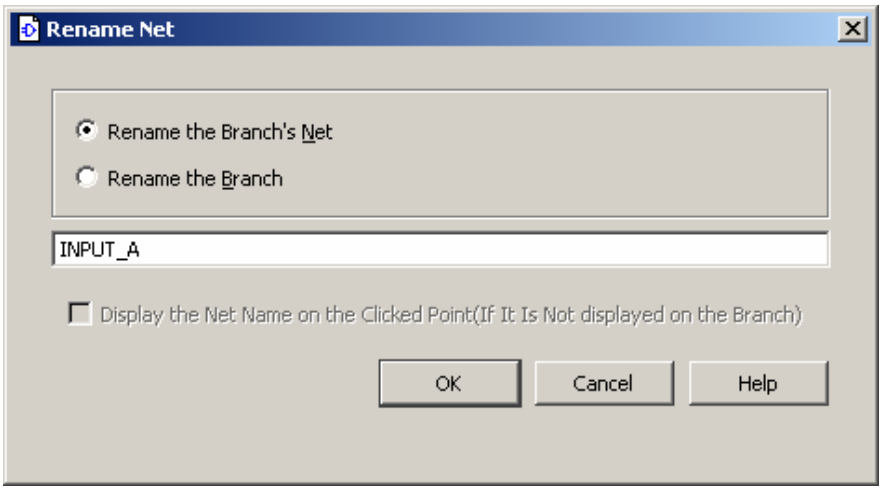


Рисунок 2.15 – Пример редактирования имени порта вывода.

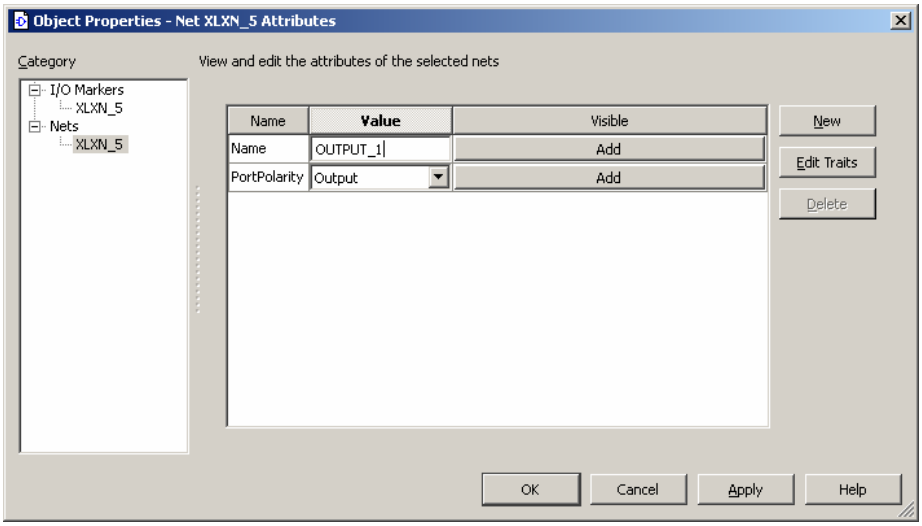


Рисунок 2.16 – Пример редактирования свойств порта вывода.

Окончательный вид принципиальной схемы представлен на рисунке 2.17.

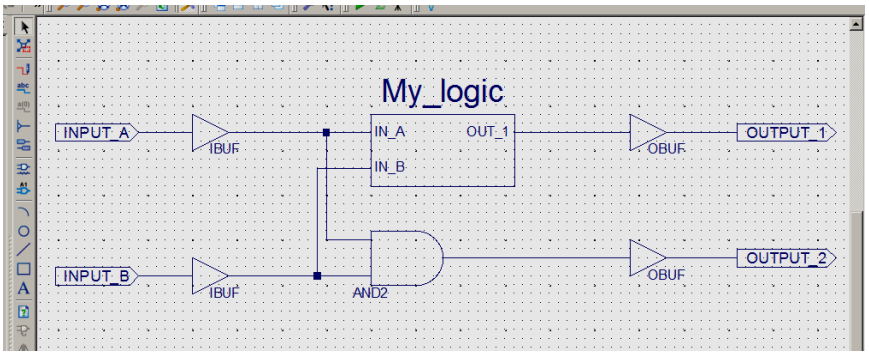


Рисунок 2.17 – Пример принципиальной схемы разработанного устройства, включающего два компонента: модуль логического описания компонента My_logic и элемент and2 из библиотеки стандартных элементов.

После редактирования принципиальной схемы следует сохранить файл с разрабо-

танной схемой устройства (модуль верхнего уровня иерархии). Для проверки нужно убедиться, что на закладке **Design** в окне модулей проекта разработанная схема ***Top_Spa_6 (Top_Spa_6.sch)*** переместилась на верхний уровень иерархии проекта, включив в себя модуль ***XLXI_XX - My_logic (My_logic.v)*** как это показано на рисунке 2.18.

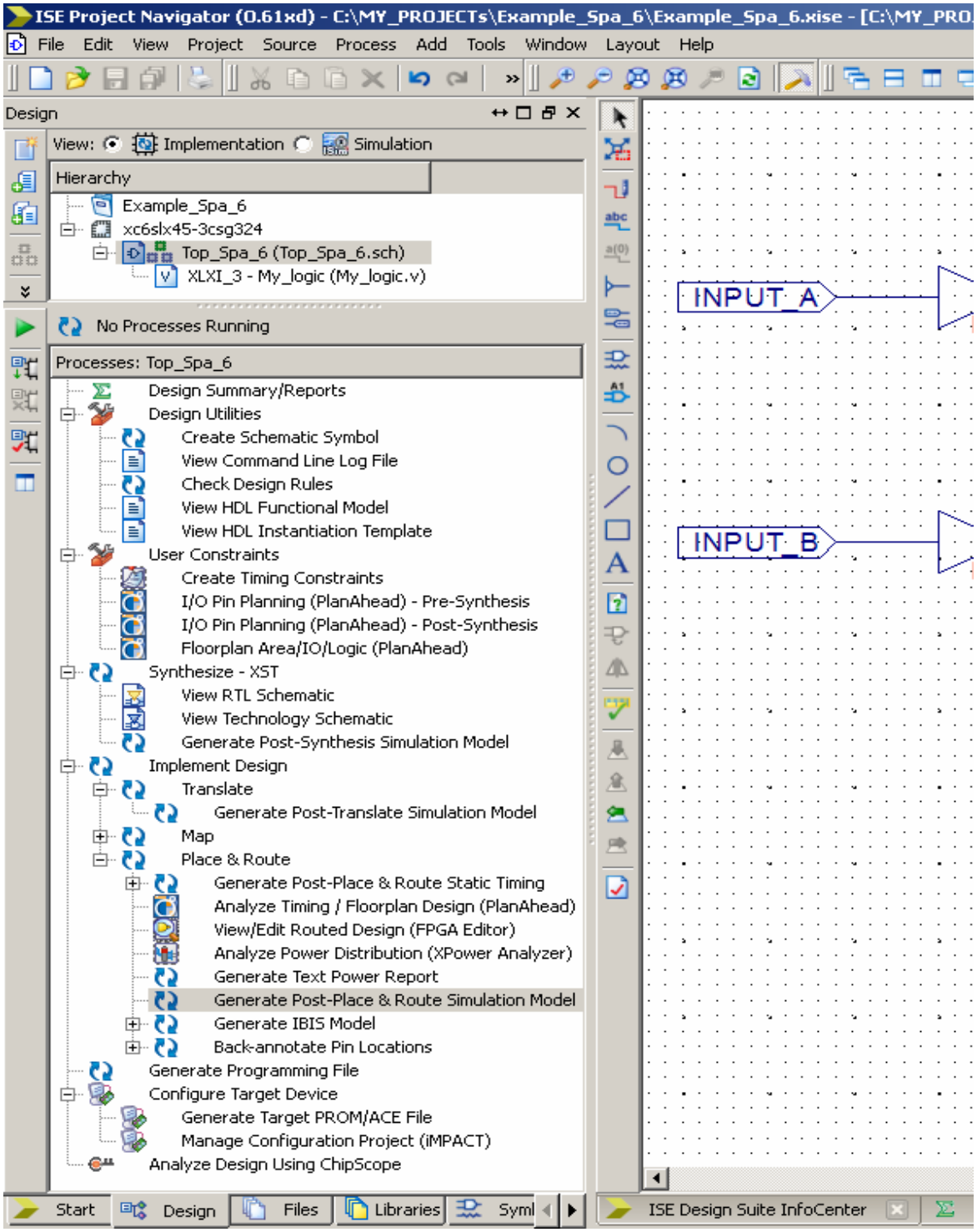


Рисунок 2.18 – Два уровня иерархии проекта (окно **Design: Implementation**). Процессы (проектные процедуры), доступные для верхнего модуля проекта ***Top_Spa_6 (Top_Spa_6.sch)*** (в окне **Process** выделен модуль **Top_Spa_6**).

2.1.4 Синтез проектируемого устройства на RTL-уровне описания и на основе стандартных элементов ПЛИС

Контекстное окно **Processes** содержит список всех проектных процедур, доступных пользователю ISE в зависимости от выделенного модуля проекта. В общем случае процесс проектирования в среде ISE заканчивается выполнением процедур трансляции главного файла проекта.

Трансляция проекта включает три основных этапа: **Synthesize** (синтез устройства), **Implement Design** (размещение устройства в выбранной микросхеме ПЛИС с использованием ее ресурсов) и **Generate Programming File** (создание бинарного файла для выполнения процедуры прошивки конфигурации разработанного устройства в ПЛИС).

Каждый из этапов трансляции может включать в себя несколько проектных процедур. На рисунке 2.18. приведен пример такого списка для модуля ***Top_Spa_6.sch.***

Пиктограммы возле отдельных названий процедур соответствуют типу приложений, которые будут запущены на выполнение при их активации. Двойными стрелками отмечены процессы, исполняемые в основном окне навигатора. Ход выполнения операций документируется в специальном текстовом файле ****.log***, и иллюстрируется сообщениями в окне консоли.

Значки галочки в зеленых кружках сигнализируют об успешном завершении операции. Желтые восклицательные знаки указывают на то, что один из вложенных процессов завершился с некритическими ошибками (предупреждениями). Критические ошибки, приводящие к прекращению процесса трансляции, отмечаются красными крестиками. Значком документа показаны файлы отчетов о полученных проектных решениях, которые будут выведены при этом в основное окно навигатора проекта. Другие пункты и пиктограммы приложений соответствуют внешним сопутствующим инструментам (утилитам), открывающимся в своем окне.

Для трансляции всего проекта достаточно запустить процесс **Generate Programming File.** При этом недостающие для его выполнения процессы будут запущены автоматически.

Параметры трансляции настраиваются из контекстных меню процессов, и их не следует изменять без четкого понимания назначения параметров и целей перенастройки. Подробнее о настройках параметров трансляции можно прочесть в книгах [2.2 и 2.4].

Порядок процедур в списке соответствует последовательному поэтапному проектированию сверху-вниз и пользователь при необходимости имеет альтернативную возможность выполнять трансляцию по шагам, контролируя результаты после каждого этапа.

В частности, на этапе размещения устройства в конкретной ПЛИС, пользователь может просмотреть как реализован проект на уровне описания цепей сигналов и сохранения их состояний в регистрах (**Register Transfer Level, RTL-уровень**). Для этого нужно вызвать из раздела списка **Synthesize –XST** процедуру **View RTL Schematic** и включить, как показано на рисунке 2.19, второй пункт в окне представления результатов.

Пример RTL-представления разработанного устройства приведен на рисунке 2.20.

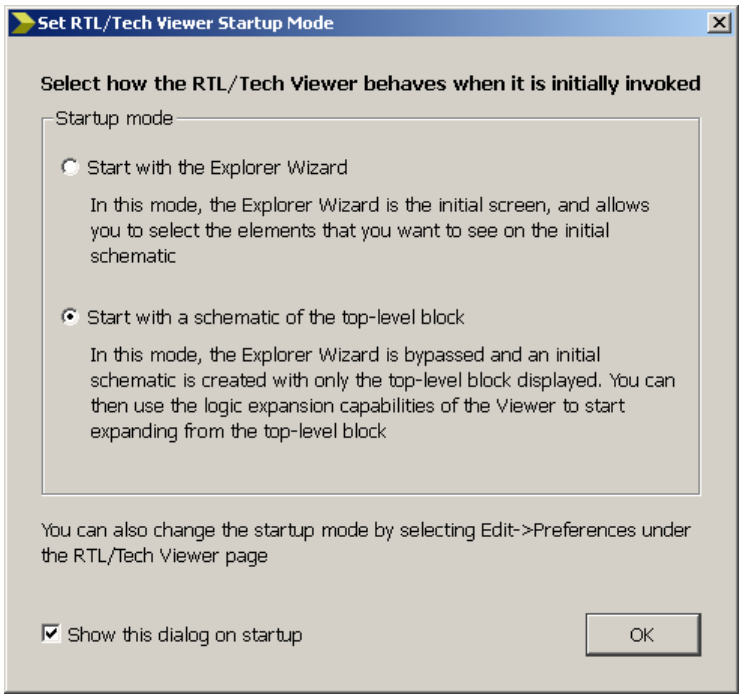


Рисунок 2.19 – Окно настройки представления результатов выполнения процедуры **Synthesize –XST - View RTL Schematic**.

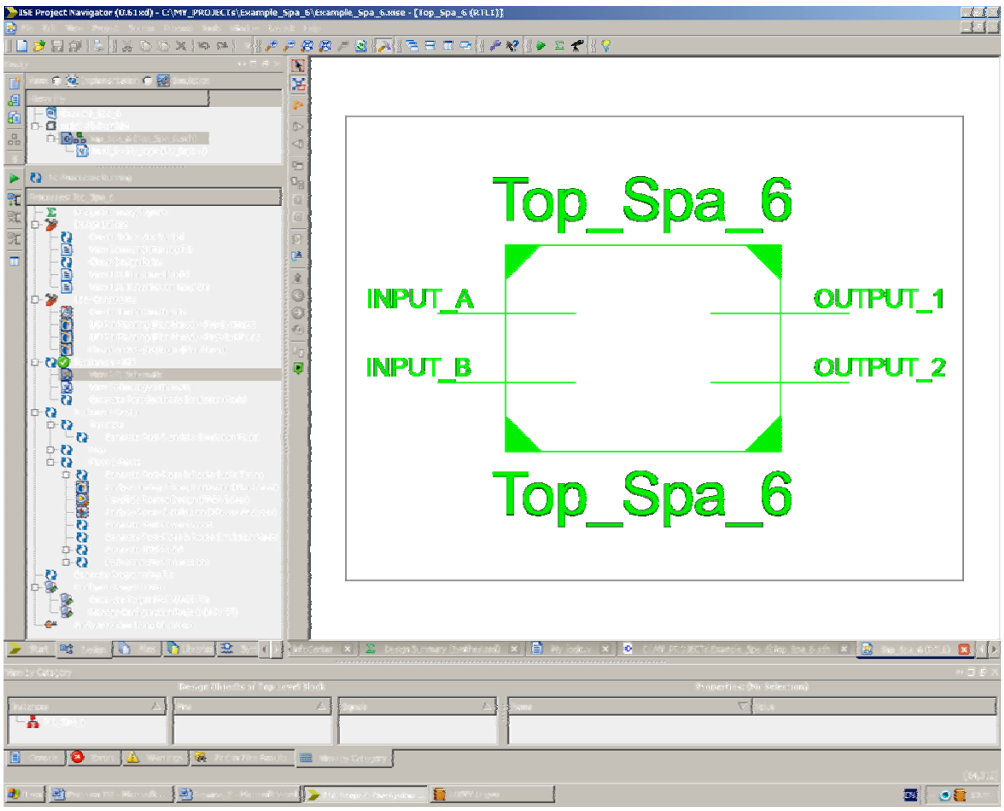


Рисунок 2.20 – Пример результата выполнения процедуры **Synthesize –XST - View RTL Schematic.** Проектируемое устройство представлено в виде модуля верхнего уровня с интерфейсными портами.

Раскрыть внутреннее содержание модуля можно двойным щелчком на его графическом символе (см. рисунок 2.21).

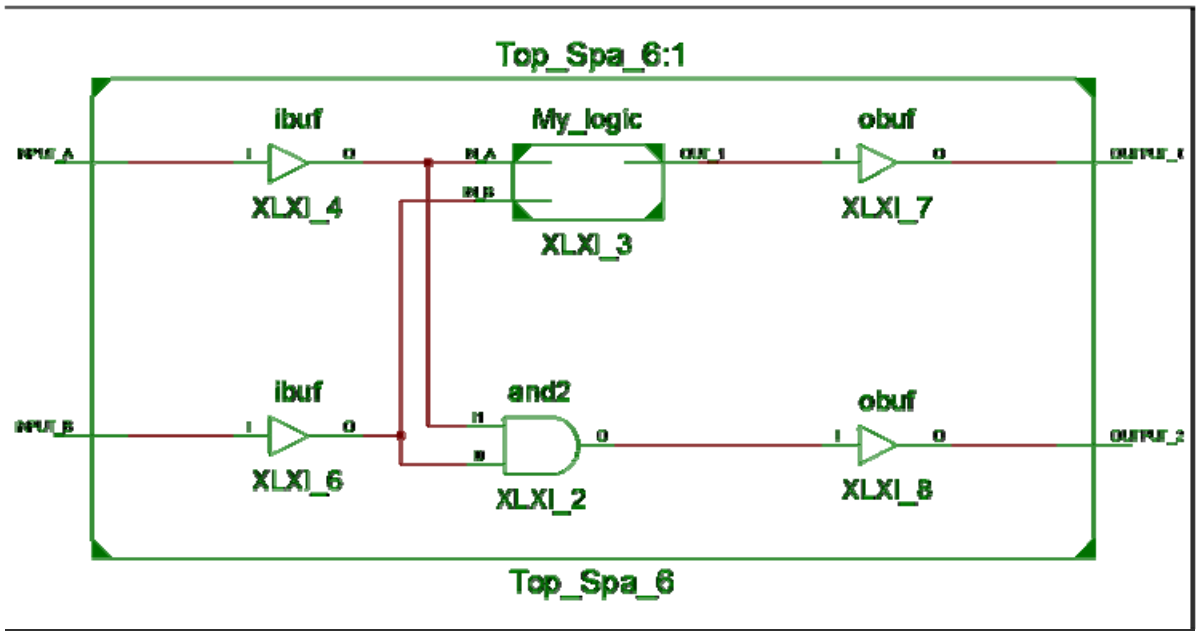


Рисунок 2.21 – Пример результата выполнения процедуры
Synthesize –XST - View RTL Schematic на нижнем уровне иерархии.

Используя процедуру **Synthesize–XST - View Technology Schematic,** можно проследить, как транслятор ISE распорядился ресурсами микросхемы, т.е. проконтролировать, как устройство собрано из стандартных элементов, содержащихся в данной микросхеме ПЛИС. Пример приведен на рисунке 2.22.

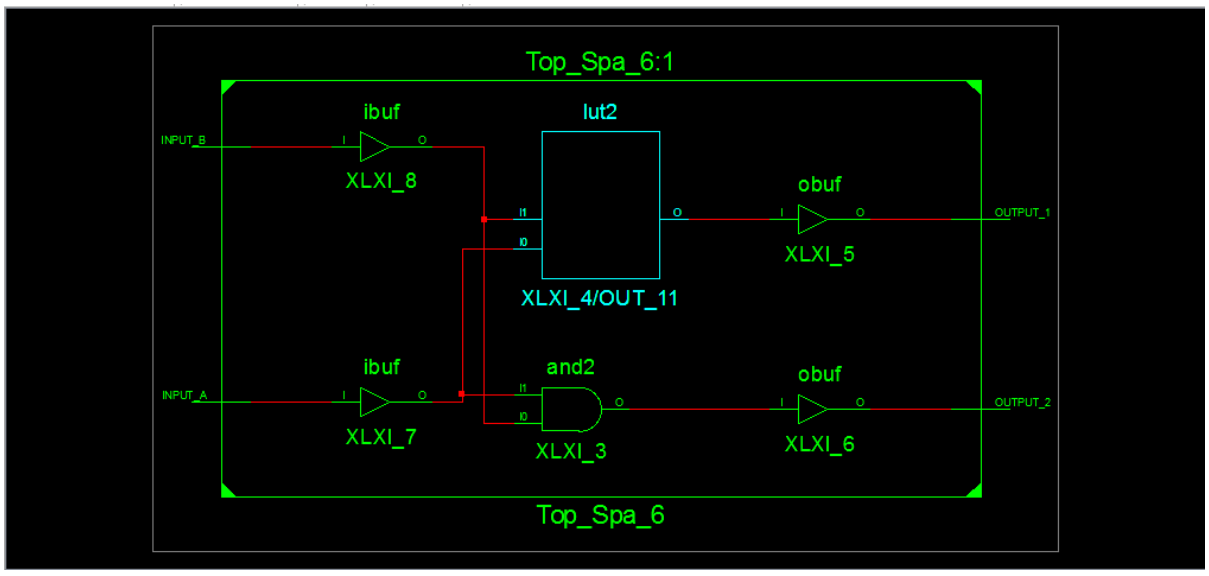


Рисунок 2.22 – Пример результата выполнения процедуры
Synthesize –XST - View Technology Schematic.

Дважды щелкнув на графическом символе логического модуля **My_logic** можно раскрыть его принципиальную схему, составленную из библиотечных элементов (см. рисунок 2.23), соответствующее логическое уравнение в совершенной дизъюнктивной форме, таблицу истинности (Look-Up Table) программируемого генератора логических функций LUT2, реализующего эту функцию (рисунок 2.24) и карту Карно (рисунок 2.25).

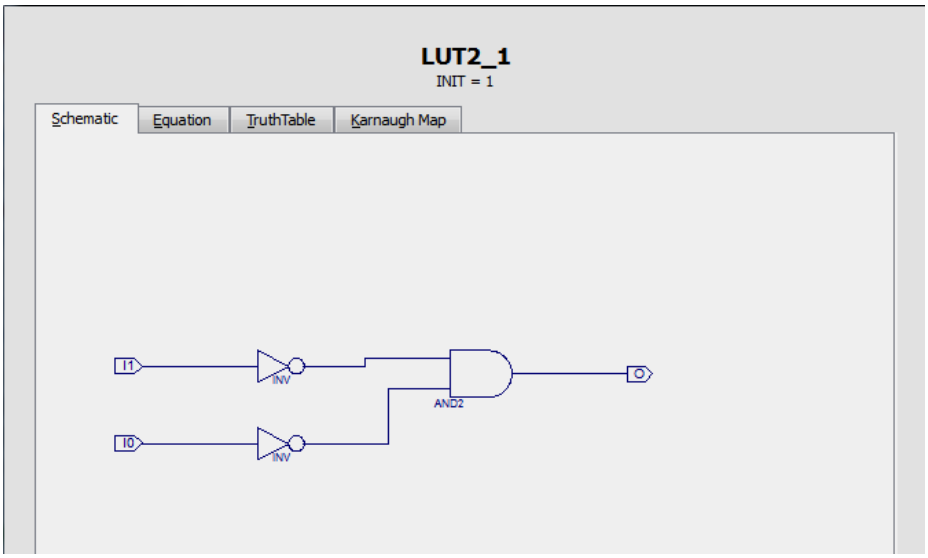


Рисунок 2.23 – Принципиальная схема логического модуля My_logic, составленного из перепрограммируемых элементов ПЛИС (т.е. синтезируемого из компонентов стандартной библиотеки).

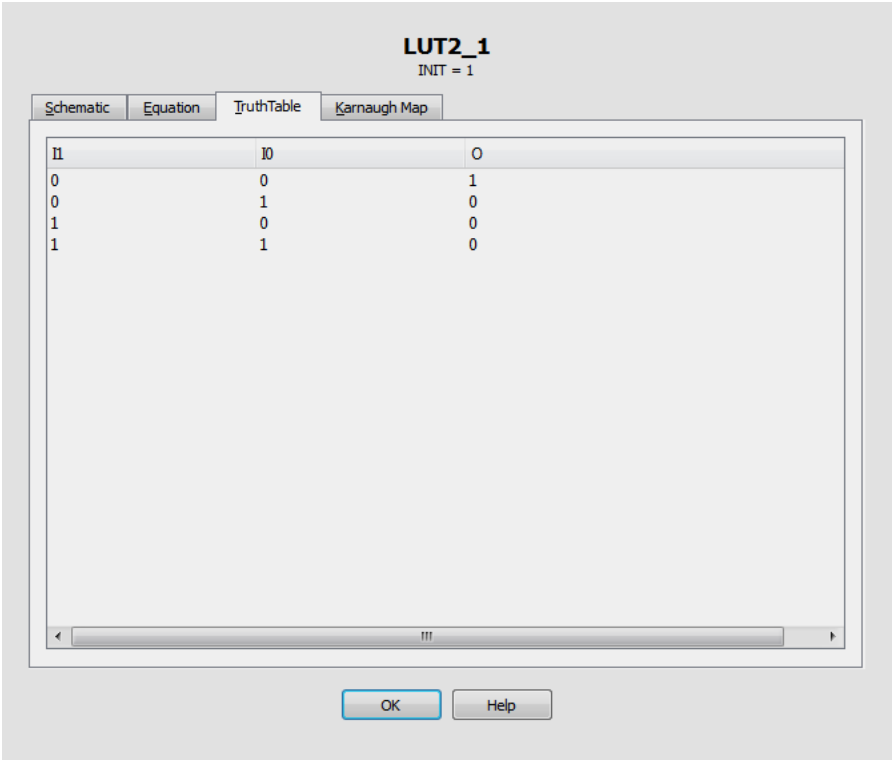


Рисунок 2.24 – Таблица истинности логического модуля **My_logic**, синтезированного генератором логических функций LUT2.

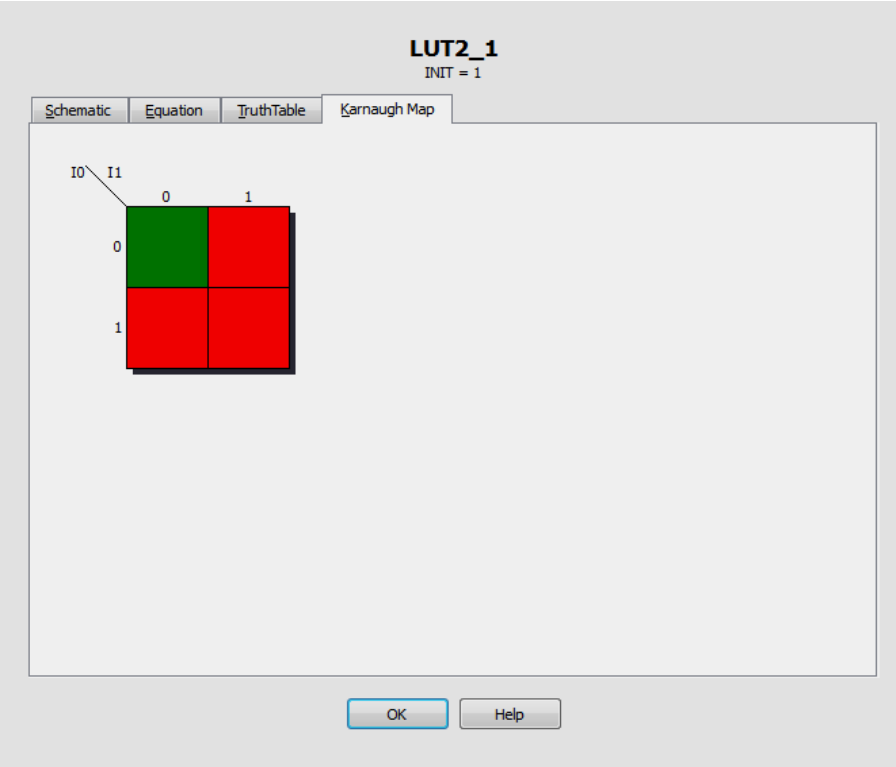


Рисунок 2.25 – Карта Карно таблицы истинности логического модуля **My_logic**.

2.1.5 Функциональное моделирование

Моделирование цифровых систем является важным шагом в маршруте их разработки. Термин компьютерное моделирование работы интегральной схемы означает – программную симуляцию поведения виртуальной модели интегральной схемы при специальном воздействии тестовых входных сигналов. Возрастание сложности проектируемых устройств заставляет разработчиков затрачивать значительные временные ресурсы на их моделирование. Моделирование широко применяется для проверки выполнения требуемых функций, оптимизации потребляемой мощности, а также для анализа влияния временных задержек сигналов на элементах схемы и оптимизации распределения тактовых сигналов внутри кристалла.

Целями моделирования могут быть также исследования алгоритмов работы проектируемого устройства и верификация характеристик, получаемых при его аппаратной реализации. В первом случае производится моделирование на верхних уровнях абстрагирования (т.е. преимущественно на поведенческом уровне, и, возможно, на RTL-уровне). Моделирование на физическом уровне призвано проверить возможность работы созданного устройства в заданных условиях эксплуатации (т.е. проверяется возможность работы на заданной тактовой частоте, с требуемыми длительностями сигналов, в заданном температурном диапазоне и т.д.).

План тестирования в идеальном случае должен обеспечивать проверку все возможных состояний проектируемого устройства (полное тестовое покрытие), что практически недостижимо для сложных цифровых устройств. Это обусловливает необходимость выработки оптимальной стратегии моделирования, что является предметом отдельных исследований при проектировании. Таким образом, моделирование представля-

ет собой специальный комплекс проектных процедур, требующий применения специальных знаний, навыков и автоматизированных инструментов верификации. Современные средства автоматизированного проектирования предлагают разработчику большую номенклатуру инструментов моделирования. Примерами могут служить пакеты программ MultiSim (Mentor Graphics) [2.7], Virtuoso (Cadence) [2.8] и VCS (Synopsys) [2.9].

САПР ISE также имеет встроенные средства моделирования – утилиту **iSim**. Подробную информацию о моделировании и верификации проектов на основе ПЛИС можно найти в учебных пособиях [2.1…2.4]. Минимальные сведения, необходимые для выполнения данного учебного проекта сводятся к следующему.

При моделировании используется подход, основанный на создании и применении специальной моделирующей программы - «испытательного стенда» (*testbench*). Для этого моделируемое (тестируемое) устройство (в англоязычной литературе - *UUT, Unit Under Test*) представляется своим синтезируемым кодом (внутренний модуль испытательного стенда), а для проверки его поведения в различных условиях создаются описания тестовых воздействий («моделирующий код», т.е. внешний модуль испытательного стенда).

Задание тестовых входных воздействий реализуется программно на основе языков HDL, в частности, на языке Verilog. Чтобы в проект добавить тестовый модуль следует вызвать из главного системного меню функцию **Project** – **New Source**…и в открывшемся диалоговом окне выбрать из списка тип вновь создаваемого модуля: ***Verilog Test Fixture*** (см. рисунок 2.26). Для унификации обычно в названии тестового модуля используется постфикс _TB (от *testbench*). В данном случае основному модулю Top_Spa_6 следует сопоставить тестовый файл Top_Spa_6 _TB.

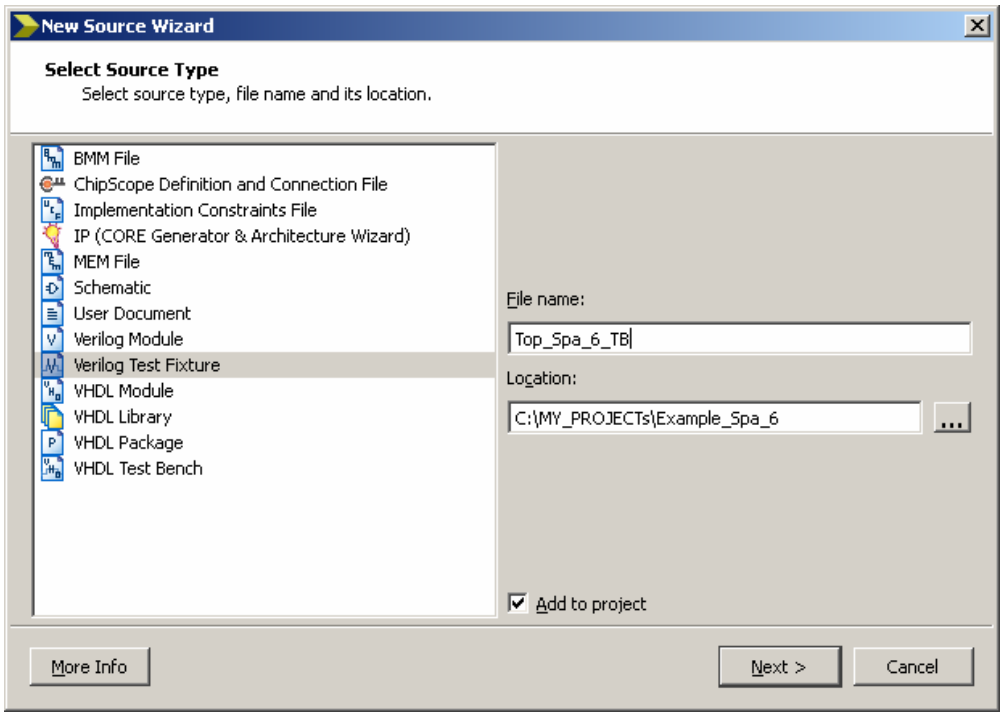


Рисунок 2.26 – Создание модуля, описывающего тестовые воздействия.

При добавлении тестового файла в проект САПР ISE также запрашивает ассоциирование этого файла с одним из имеющихся модулей (см. рисунок 2.27). Запрос на ассоциирование нужен для того, чтобы сгенерировать шаблон теста испытательного стенда с автоматической привязкой к интерфейсу (т.е. к портам) тестируемого модуля.

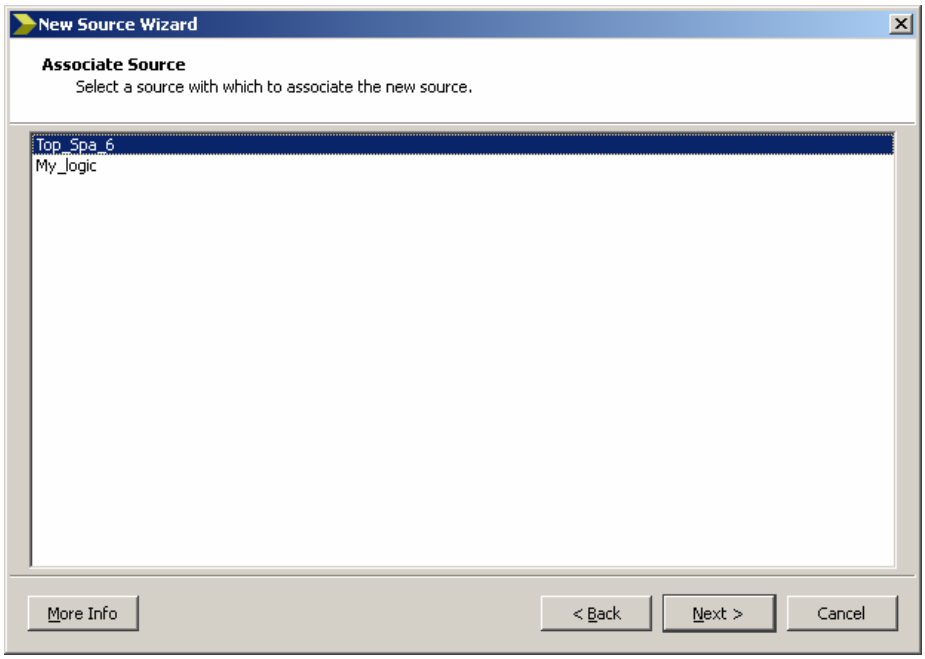


Рисунок 2.27 – Привязка модуля описания тестовых воздействий к синтезируемому модулю проекта.

Результат генерации шаблона тестового файла для верхнего модуля проекта ***Top_Spa_6.sch*** показан на рисунке 2.28.

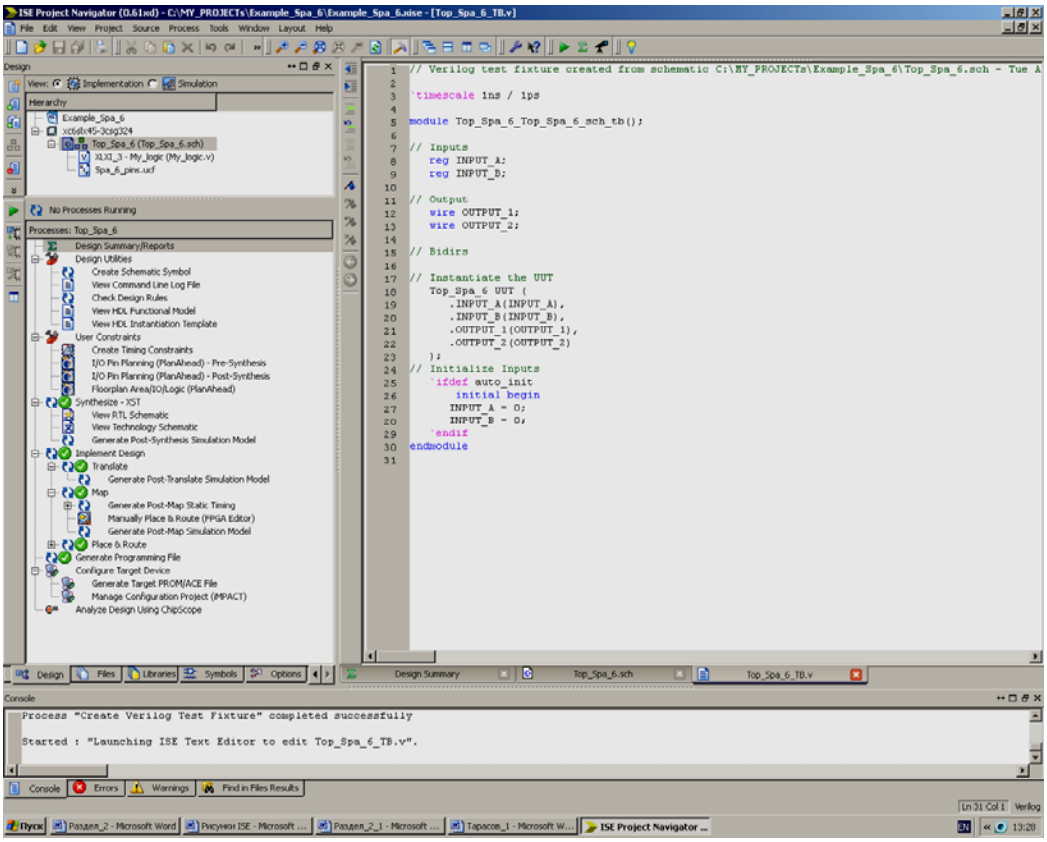


Рисунок 2.28 – Пример генерации шаблона модуля описания тестовых воздействий (файл ***Top_Spa_6_TB.v***).

При генерации поведенческого описания тестового модуля необходимо строго придерживаться правил написания кодов на языке HDL, в данном случае – на языке Verilog. Основы проектирования систем с использованием языка поведенческого описания аппаратуры Verilog достаточно полно изложены в учебных пособиях [2.10…2.12]. Для разработки тестового файла в данном проекте необходимо принять во внимание следующее.

1. В Verilog существуют два основных *типа данных (сигналов)*: *цепи* (`net`) и *переменные* (`variable`). Принципиальное различие между ними состоит в поведении при моделировании. **Цепи постоянно отслеживаются**, и их состояние **обновляется немедленно** после изменения состояния сигналов, от которых зависит состояние цепи. Состояние **переменных обновляется только внутри специальных *процедурных блоков*,** вносимых разработчиком в текст модуля на Verilog.
2. Основным типом *цепей* является тип, идентифицируемый ключевым словом **`wire`**. Все сигналы по умолчанию имеют тип **`wire`**.
3. Основным типом *переменных* в Verilog является тип, объявляемый ключевым словом **`reg`**. Это объект, который сохраняет записанное в него значение, и изменяет его только в явно определенные моменты. Для задания значений однобитовых переменных типа **`reg`** используют формат: 1’b1 или 1’b0 – (двоичное число 1 или 0).
4. Объектам, имеющим тип **`reg,`** значения могут присваиваться внутри процедурных блоков **`initial`** и **`always`**.
5. Модули, описанные на языке Verilog, обязательно имеют объявление портов – внешних выводов (т.е. интерфейса) модуля, функциональное поведение которого описано в теле этого модуля. Структура модуля на языке Verilog:

```
module <имя> ([параметры] <объявления портов>)
[объявления локальных сигналов и переменных]
<синтезируемые конструкции>
Endmodule
```

6. Для создания проектов из нескольких модулей требуется их объединение. Объединения описываются с применением конструкции **component instantiation** (конкретная реализация компонента): ранее созданные компоненты указываются в тексте модуля верхнего уровня с перечислением сигналов, которые необходимо подключить к их портам. Например, запись

```
Top_Spa_6 UUT .In_a(Sign_c),
```

   означает, что порт `In_a` внутреннего модуля с конкретным именем UUT, создаваемого на основе модуля Top_Spa_6 (имеющегося в проекте прототипа с таким же внутренним содержанием и интерфейсом) должен быть подключен к цепи `Sign_c` внешнего модуля.

При разработке тестовых модулей используется распространенный прием проектирования, состоящий в том, что сложный проект представляет собой иерархическую структуру, составленную из вложенных модулей, причем тестовый модуль является верхним. В частном случае, как показано на рисунке 2.29, на котором приведен листинг шаблона тестового модуля, при выполнении команды ***Verilog Test Fixture***, программа сама сгенерировала внешний тестовый модуль `Top_Spa_6_Top_Spa_6_sch_tb.`

```
//Verilog·      test·      fixture·      created·      from·      schematic·
//C:\MY_PROJECTs\Example_Spa_6\Top_Spa_6.sch·-·Tue·Aug·16·13:20:46·2011¶
¶
`timescale·1ns·/·1ps¶
¶
module·Top_Spa_6_Top_Spa_6_sch_tb();¶
¶
//·Inputs¶
reg·INPUT_A;¶
   reg·INPUT_B;¶
¶
//·Output¶
   wire·OUTPUT_1;¶
   wire·OUTPUT_2;¶
¶
//·Bidirs¶
¶
//·Instantiate·the·UUT¶
   Top_Spa_6·UUT·(¶
  →     →   .INPUT_A(INPUT_A),·¶
  →     →   .INPUT_B(INPUT_B),·¶
  →     →   .OUTPUT_1(OUTPUT_1),·¶
  →     →   .OUTPUT_2(OUTPUT_2)¶
···);¶
//·Initialize·Inputs¶
···`ifdef·auto_init¶
·······initial·begin¶
  →     →   INPUT_A·=·0;¶
  →     →   INPUT_B·=·0;¶
    end¶
    `endif¶
endmodule¤
```

Рисунок 2.29 – Листинг шаблона модуля описания тестовых воздействий (файл ***Top_Spa_6_TB.v***).

В этот модуль вложен модуль с тестируемым устройством UUT созданный из исходного компонента `Top_Spa_6`, имеющего те же свойства и интерфейс, что и `Top_Spa_6.sch`

Шаблон модуля описания тестовых воздействий содержит следующие элементы:

- Директиву трансляции:

```
`timescale 1ns / 1ps –
```

Эта директива указывает транслятору, что нужно задать условное (программное) время моделирования 1 ns, причем шаг дискретизации по времени должен составлять 1 ps.

- Модуль верхнего уровня иерархии теста:

```
module Top_Spa_6_Top_Spa_6_sch_tb();
….
Endmodule
```

- Определения типов и обозначений входных и выходных данных (сигналов) тестового модуля:

```
// Inputs
   reg INPUT_A;
   reg INPUT_B;

// Output
   wire OUTPUT_1;
   wire OUTPUT_2;

// Bidirs
```

- Модуль тестируемого устройства:

```
// Instantiate the UUT
   Top_Spa_6 UUT (
          .INPUT_A(INPUT_A),
          .INPUT_B(INPUT_B),
          .OUTPUT_1(OUTPUT_1),
          .OUTPUT_2(OUTPUT_2)
   );
```

- Директивы условной трансляции, выполняющие функции инициализации (задания начальных значений) переменным модуля верхнего уровня:

```
// Initialize Inputs
   `ifdef auto_init
       initial begin
          INPUT_A = 0;
          INPUT_B = 0;
       end
   `endif
```

Следует обратить внимание на то, что в автоматически созданном шаблоне имена портов модуля верхнего уровня совпадают с именами портов подключаемого компонента – модуля тестируемого устройства UUT. Согласно синтаксису языка, такое совпадение допустимо и не является ошибкой, поскольку имена сигналов внутреннего модуля могут быть произвольными и не должны влиять на работу внешнего модуля.

Для данного проекта полное тестовое покрытие заключается в отслеживании сигналов на выходах OUTPUT_1 и OUTPUT_2 модуля UUT при всех возможных состояниях его входов INPUT_A и INPUT_B. Примем следующий план тестирования:

1. В начальный момент времени состояния сигналов не определены.
2. Через 10 единиц условного времени осуществятся инициализация начального состояния: входные сигналы принимают значения 1.
3. В процессе тестирования сигнал, поступающий на вход INPUT_A тестируемого модуля является импульсным с периодом 100 единиц условно времени (т.е. каждые 50 единиц времени изменяет свое состояние на противоположное);
4. Сигнал, поступающий на вход INPUT_B тестируемого модуля один раз за время теста (через 200 единиц условно времени после инициализации) принимает значение 0 и через 200 единиц времени снова принимает значение 1.

Функциональная схема, поясняющая план тестирования, принятые обозначения и организацию тестового файла приведена на рисунке 2.30.

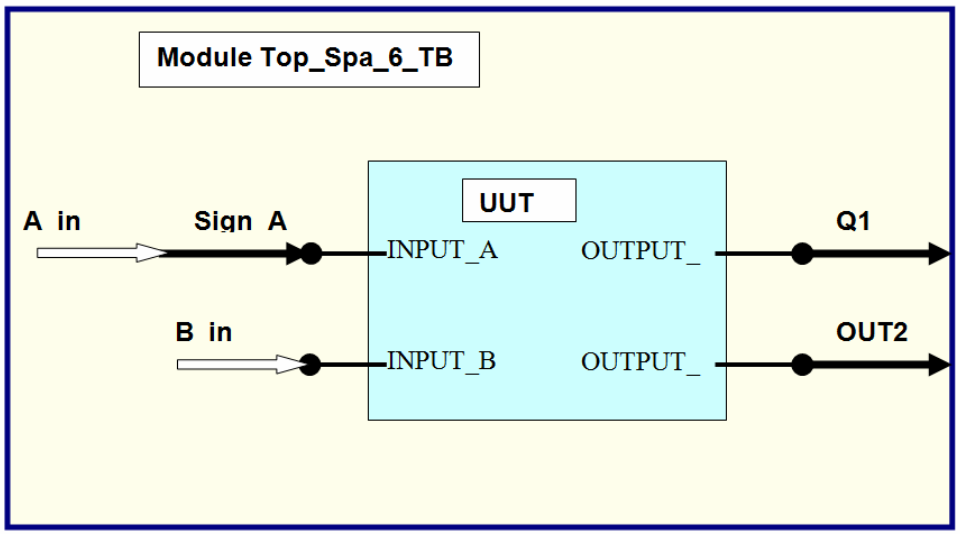


Рисунок 2.30 – Функциональная схема тестовых воздействий проекта ***Top_Spa_6_TB.v***.

На схеме обозначены:

- INPUT_A, INPUT_B, OUTPUT_1, OUTPUT_2 - порты тестируемого модуля ***Top_Spa_6.sch***;
- `A_in` и `B_in` – сигналы, которые должны быть сгенерированы тестовым модулем (имеют тип данных `reg`, т.к. их состояние должно принудительно задаваться извне в процессе тестирования);
- `Sign_A` – сигнал, имеющий тип `wire`, значение которого должно меняться периодически в процессе моделирования и который должен быть подключен к входному порту INPUT_A;
- `Q1` и `OUT2` – сигналы на выходе тестового модуля (поскольку их состояние должно изменяться в соответствии с изменениями входных воздействий, то они также имеют тип `wire`).

В соответствии с принятым планом тестирования в текст шаблона, приведенный в листинге на рисунке 3.31 необходимо внести следующие изменения:

В таблице 2.4. приведены ожидаемые состояния выходных сигналов проектируемого устройства в соответствующие интервалы времени.

Таблица 2.4 – Ожидаемые состояния выходных сигналов в соответствии с принятым планом тестирования.

Интервал времени, ps	0…10	10…50	50…100	100…150	150..200	200…210	210…250	250…300	300…350	350…400	400…410
A_in	x	1	0	1	0	1	1	0	1	0	0
B_in	x	0	0	0	0	0	1	1	1	1	1
Q1	x	0	1	0	1	0	0	0	0	0	0
OUT2	x	0	0	0	0	0	1	0	1	0	1

```
`timescale 1ns / 1ps

module Top_Spa_6_Top_Spa_6_sch_tb();

// Inputs
   reg A_in; // **
   reg B_in; // **
   wire Sign_A; // **

   assign Sign_A = A_in;  // **

// Output
   wire Q1; // **
wire OUT2; // **

// Bidirs


// Instantiate the UUT
Top_Spa_6 UUT (
            .INPUT_A(Sign_A), // **
            .INPUT_B(B_in), // **
            .OUTPUT_1(Q1), // **
            .OUTPUT_2(OUT2) // **
);
// Initialize Inputs

initial // **
      begin // **
      #10; // **
      A_in = 1'b1; // **
      B_in = 1'b0; // **
      #200; // **
      B_in = 1'b1; // **
      #200; // **
      B_in = 1'b0; // **
      end // **
// Simulation
      always // = always #50 A_in = ~ (A_in); // **
      begin // **
      #50; // **
      A_in = ~ (A_in); // **
      end // **

   endmodule
```

Знаками // ** отмечены строки, которые добавлены в исходный модуль шаблона.

Рисунок 2.31 – Листинг модуля описания тестовых воздействий после внесения изменений в соответствии с принятым планом тестирования и функциональной схемой тестового модуля (исправленный файл ***Top_Spa_6_TB.v***).

После того как тестовый файл разработан, следует сохранить его в проекте. Для переключения режимов работы навигатора проектов между исходными файлами про-

екта и тестовыми модулями следует использовать флаги переключения **Implementation – Simulation** окна модулей проекта как это показано на рисунке 2.32.

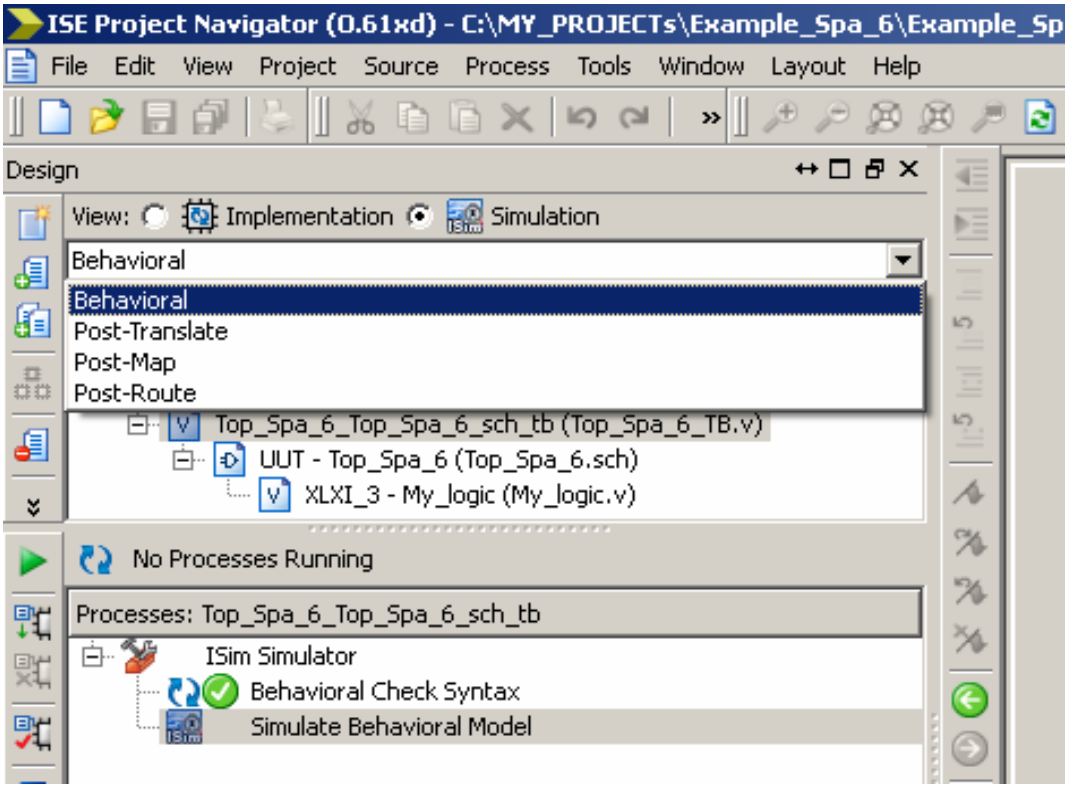


Рисунок 2.32 – Флаги **Implementation – Simulation** переключения режимов работы окна навигатора проекта и список настроек работы программы моделирования.

На этом же рисунке выпадающий список содержит перечень настройки режимов работы программы симуляции.

Настройка режимов симуляции позволяет проводить в САПР ПЛИС ISE моделирование не только на поведенческом (Behavioral), но и на физическом уровне. В этом случае задержки распространения сигналов не принимаются равными тем, которые указаны в поведенческом описании, а рассчитываются, исходя из физических моделей компонентов ПЛИС, размещения на кристалле и трассировки конкретного проекта. Для моделирования в таком режиме необходимо в выпадающем списке выбрать соответствующий режим **Post -**… перед запуском моделирования.

В данном случае следует оставить режим **Behavioral** (проверка функционального поведения устройства).

Отредактированный файл тестового модуля следует проверить на наличие синтаксических ошибок. Для этого нужно перейти на вкладку **Simulation** окна исходных модулей навигатора проекта, выделить тестовый модуль (в данном случае – модуль ***Top_Spa_6_Top_Spa_6_sch_tb***) и запустить на исполнение процесс **Behavioral Check Syntax** из окна **Process** (см. рисунок 2.33).

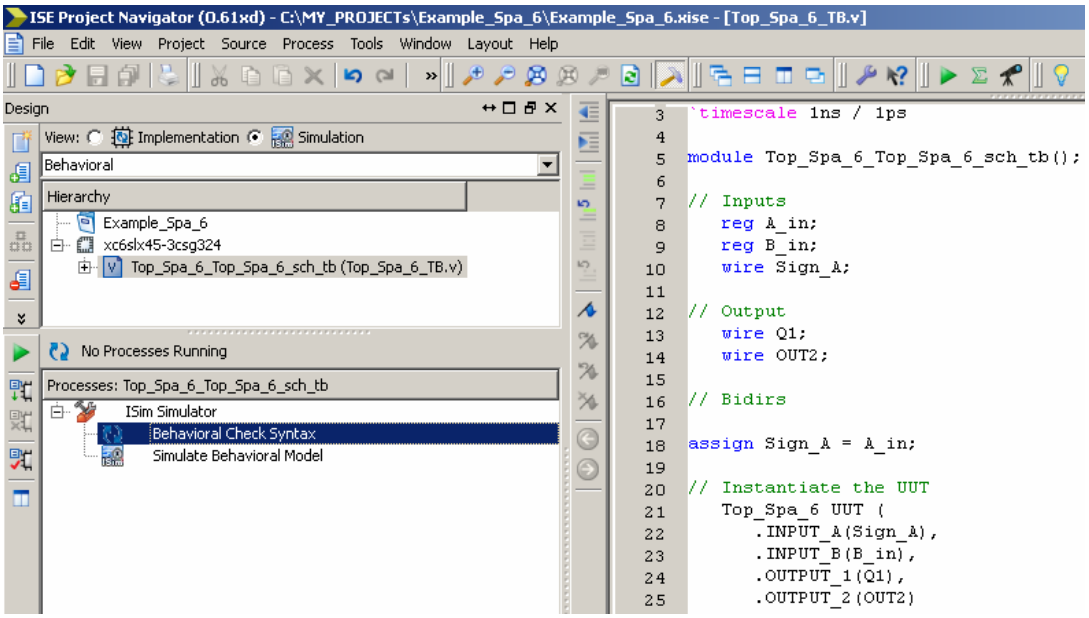


Рисунок 2.33 – Проверка синтаксиса файла тестовых воздействий (испытательного стенда).

Ошибки следует исправить, сохранить исправленный файл, и вновь проверить синтаксис. Если ошибок нет, то можно приступить к моделированию. Для этого в окне процессов нужно запустить на исполнение программу моделирования, нажав дважды на строку **Simulate Behavioral Model**. В результате начнет работать специальная программа моделирования **iSim** и по завершении теста откроется основное окно программы (см. рисунок 2.34).

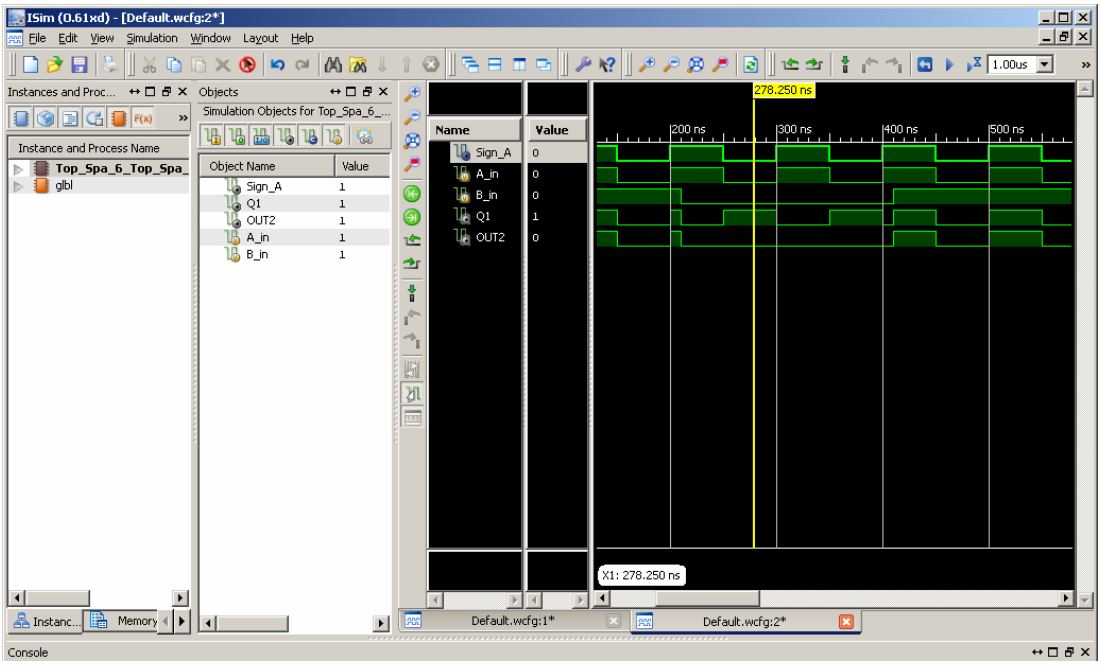


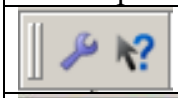
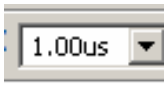
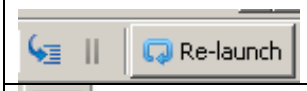
Рисунок 2.34 – Интерфейс программы моделирования iSim.

Инструмент моделирования **iSim,** в свою очередь содержит следующие элементы управления: главное меню, палитру инструментов, окно имен классов и процессов

проекта (**Instance and Process Name**), окно объектов моделирования.(**Object Name**) и окно временной диаграммы с результатами моделирования.

Функции программы симулятора снабжены контекстными всплывающими подсказками. Основные функции палитры инструментов, необходимые для анализа временной диаграммы перечислены в таблице 2.5.

Таблица 2.5 – Основные кнопки панели инструментов для работы с временной диаграммой программы **iSim**.

Пиктограммы	Назначение элементов управления
	вызов справочной системы, информация о выделенном объекте
	настройка размещения окон программы
	отображение текущего вида диаграммы – изменение масштаба, отображение всей диаграммы, отображение фрагмента между маркерами
	перерисовать все окна
	перейти к предыдущему или следующему моменту изменения состояния диаграммы, добавить маркер, сделать активным предыдущий или следующий маркер
  	управление запуском и остановкой временной диаграммы: сброс симуляции, постоянная работа, выполнить моделирование до момента, указанного на панели инструментов, задание общего времени моделирования, выполнить по шагам, остановить моделирование, перезапустить сначала
	перейти к началу, перейти к концу
	поменять местами маркеры
	показывать моменты изменения состояния (переходы)
	поместить скользящую шкалу

При работе с данным приложением следует иметь в виду, что при необходимости внести коррективы в тестовый файл, например, для изменения времени событий, следует в окне **Instance and Process Name** щелчком мыши на названии тестового файла открыть редактор кода, внести необходимые изменения, запомнить файл, снова проверить его синтаксис. После этого в окне документов нужно открыть временную диаграмму перезапустить ее симулирование. При закрытии программы симулирования и переходе к интерфейсу навигатора проекта нужно сохранить в проекте новую редакцию тестового файла.

Другая последовательность действий при необходимости внесения изменений в файл тестовых воздействий: закрыть программу **iSim**, выполнить редактирование тестового файла и проверку его синтаксиса, а затем снова вызвать процесс **Simulate Behavioral Model**.

Полученную при моделировании временную диаграмму изменения состояний сигналов (цепей и переменных) можно сохранить в проекте для последующего просмотра. Соответствующий файл имеет расширение **.wcfg**.

2.1.6 Задание проектных ограничений

Для реализации разработки в конкретном типе микросхемы ПЛИС необходимо, в частности, указать САПР, к каким именно выводам микросхемы необходимо подключить имеющиеся в проекте цепи. При этом требуется четко понимать, каким образом к элементам проектируемой системы, внешним по отношению к ПЛИС, будут подключены входные и выходные сигналы микросхемы ПЛИС. В общем случае для этого следует воспользоваться данными о типах корпусов, назначении и характеристиках выводов микросхемы, содержащимися в ее описании (информация из Data Sheet микросхемы), а также данными о разрабатываемой системе в целом. В данном проекте эти сведения конкретизированы тем, что микросхема ПЛИС имеет определенный тип корпуса (корпус CSG324 с 324 шариковыми выводами) и смонтирована на демонстрационной плате ATLYS (см. [2.6], а также рисунки 2.35 и 2.36).

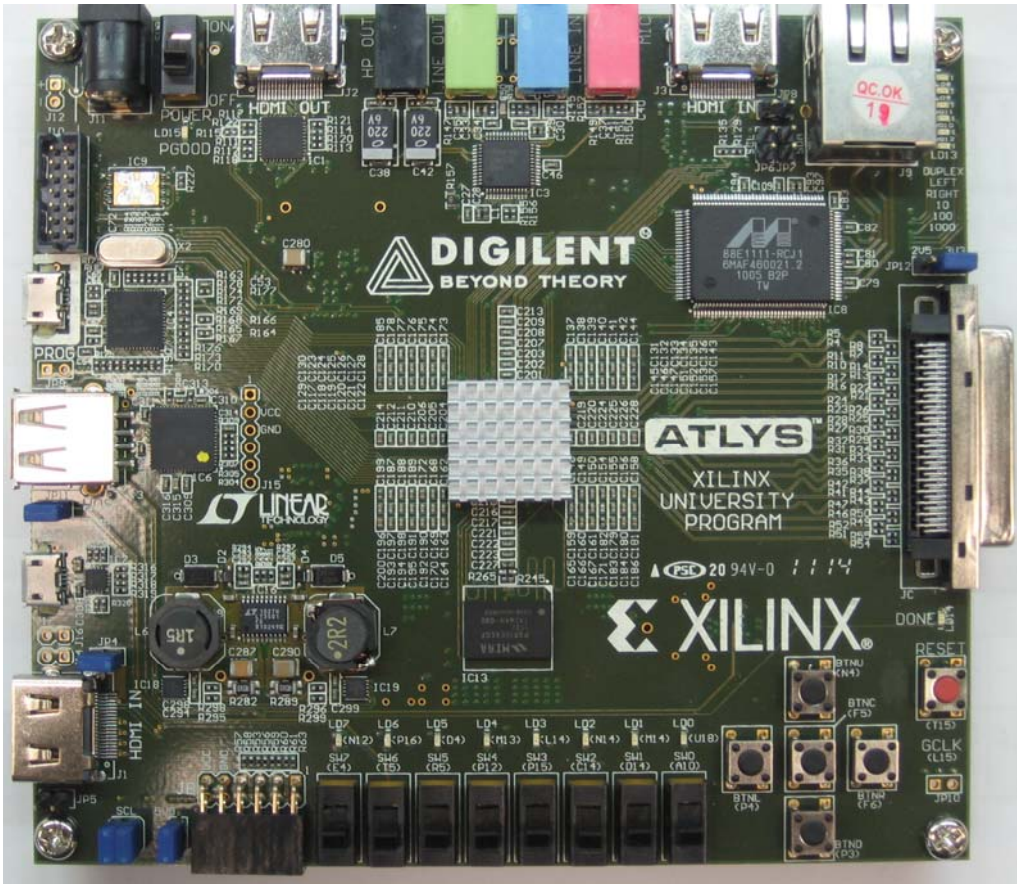


Рисунок 2.35 – Внешний вид демонстрационной платы ATLYS.

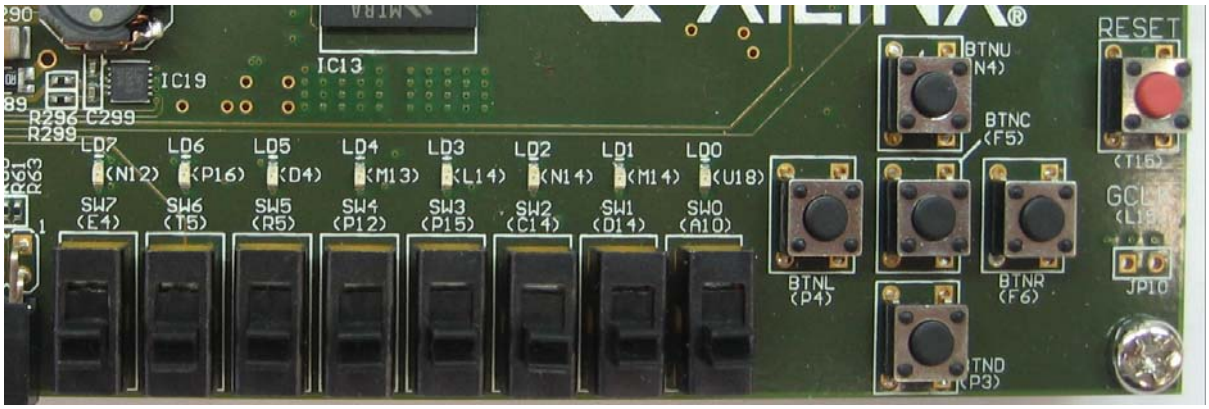


Рисунок 2.36 – Расположение ползунковых переключателей, светодиодных индикаторов и кнопок на демонстрационной плате ATLYS.

Из документации на плату ATLYS, представляющую собой по отношению к ПЛИС систему верхнего уровня, и, учитывая требования технического задания, определяем, что входные сигналы должны быть сформированы любой парой ползунковых переключателей, подключенных к выводам микросхемы в соответствии с таблицей 2.6. Для вывода выходных сигналов следует использовать светодиодные индикаторы, также подключенные к определенным выводам микросхемы и включающиеся высоким уровнем сигнала (стандартная ТТЛ-логика).

Таблица 2.6 – Назначение и номера выводов ПЛИС на плате ATLYS.

Ползунковые переключатели и соответствующие выводы ПЛИС:								
плата ATLYS	**SW0**	**SW1**	**SW2**	**SW3**	**SW4**	**SW5**	**SW6**	**SW7**
микросхема CSG324	**A10**	**D14**	**C14**	**P15**	**P12**	**R5**	**T5**	**E4**
Светодиодные индикаторы и соответствующие выводы ПЛИС								
плата ATLYS	**LD0**	**LD1**	**LD2**	**LD3**	**LD4**	**LD5**	**LD6**	**LD7**
микросхема CSG324	**U18**	**M14**	**N14**	**L14**	**M13**	**D4**	**P16**	**N12**

Для задания подключений выводов микросхемы ПЛИС к проектируемому устройству в ISE служит текстовый ***файл проектных ограничений*** (***Implementation Constraint File***), представляющий собой строки директив, интерпретируемых при трансляции проекта. Обязательными являются указания соответствия сигналов и номеров выводов. Неупомянутые параметры выводов задаются транслятором по умолчанию.

Этот файл можно добавить в проект двумя способами:

1) С помощью мастера создания новых проектных модулей (Главное меню – **Projects** – **New Source**…), и последующего вызова из контекстного меню процедуры **Edit Constraints (Text)**, открывающий соответствующий текстовый файл для редактирования в текстовом редакторе ISE.

2) Из списка меню **Process** путем выбора пункта **User Constraints** и запуска одного из процессов **I/O Pin Planning (PlanAhead)** или **Floorplan Area/IO/Logic (PlanAhead)** в списке процессов, доступных для модуля верхнего уровня.

Во втором случае из САПР ISE 13.2 будет запущено на исполнение специальное вспомогательное приложение **PlanAhead**. Подробное описание порядка работы с инст-

рументами **PlanAhead** можно найти в справочной системе ISE [2.1], а также в учебных пособиях [2.2…2.4].

Для упрощения работы в учебных целях целесообразно воспользоваться первым способом. Пример приведен на рисунке 2.37.

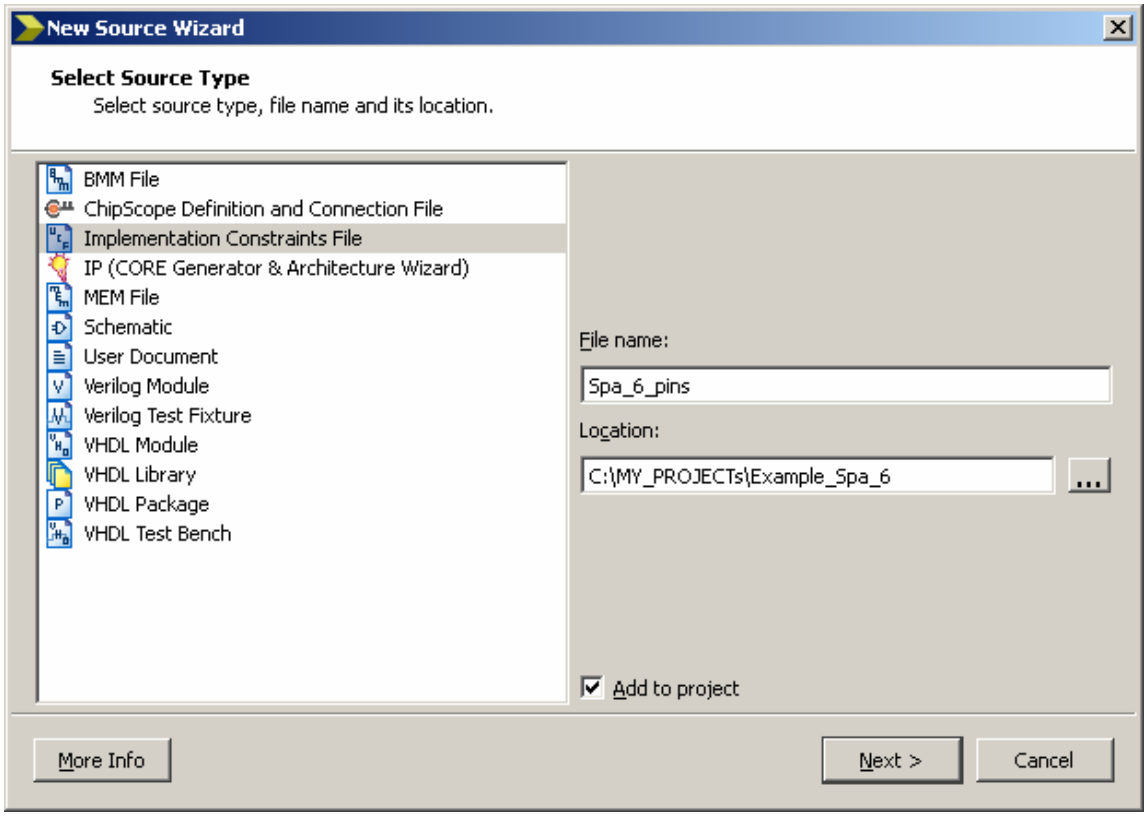


Рисунок 2.37 – Создание файла проектных ограничений для назначения сигналов и выводов микросхемы ПЛИС.

В результате в окно модулей проекта будет помещен файл ***Spa_6_pins.ucf***, который следует открыть для редактирования в окне документов для чего в окне **Process** следует дважды щелкнуть мышью на строчке **Edit Constraints (Text)**. На этом этапе проектирования нужно внести в этот файл директивы назначения выводов и их параметров. Сюда относятся данные о реализуемых аппаратно внутри ПЛИС задержках сигналов на выводах, уровнях логических сигналов, ограничениях токов потребления выходов, нагрузочных способностях выводов, внутреннее соединение через резисторы с шиной питания и другие. Подробно информация о синтаксисе директив и возможных параметрах логических и тактовых входов и выходов приведена в справочной системе ISE [2.1]. Пример текстового файла проектных ограничений для данного конкретного проекта:

```
# назначение параметров входных цепей
# сигнал INPUT_A соединен в устройстве с выводом ПЛИС A10
# (вывод на плате подключен к SW0):
NET "INPUT_A" LOC = A10;
```

```
NET "INPUT_B" LOC = D14; //INPUT_B соединен с D14 (подключен к SW1)
# назначение параметров выходных цепей
# сигнал OUTPUT_1 соединен в устройстве с выводом ПЛИС U18 (вывод на
# плате подключен к LD0):
NET "OUTPUT_1" LOC = U18;
NET "OUTPUT_2" LOC = M14; // OUTPUT_2 соединен с M14 (подключен к LD1)
```

Пример с открытого окна тестового редактора с соответствующим файлом представлен на рисунке 2.38.

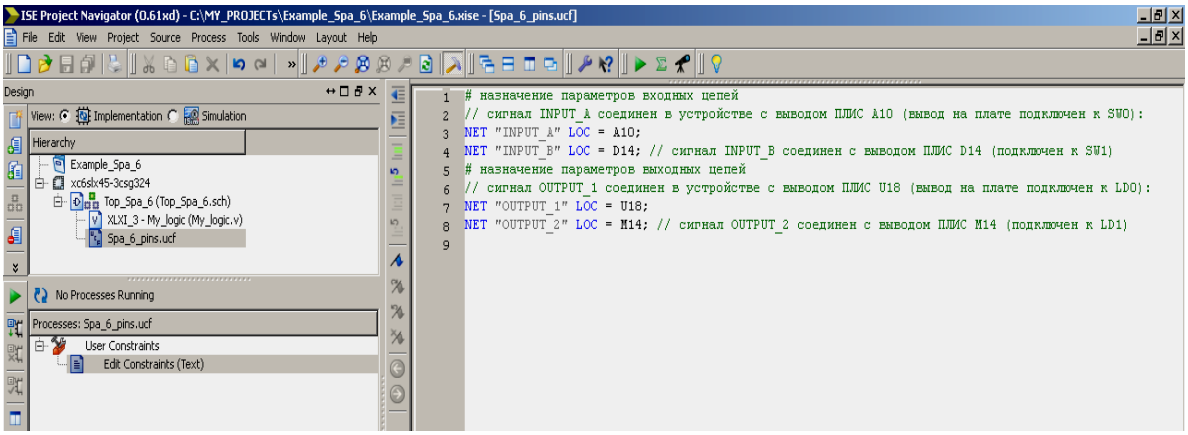


Рисунок 2.38 – Редактирование файла проектных ограничений
для назначения сигналов и выводов микросхемы ПЛИС.

После того, как все файл с назначенными выводами ПЛИС отредактирован, его следует сохранить в проекте. После этого можно проводить тестирование устройств с учетом распределения внутренних ресурсов ПЛИС и задержек прохождения сигналов при таком распределении ресурсов.

Последним этапом трансляции проекта и является этап подготовки битового файла для программирования (прошивки) микросхемы ПЛИС. После задания пользовательских ограничений становится доступной для выполнения процедура **Generate Programming File**.

Для создания файла прошивки следует выделить модуль верхнего уровня проекта (в данном случае модуль ***Top_Spa_6 (Top_Spa_6.sch)*** и на вкладке **Design** запустить на исполнение процесс **Generate Programming File.** Ход трансляции будет показан в консоли. При этом знаки «галочки» в зеленых кругах напротив каждого процесса указывают на его успешное завершение, желтые восклицательные знаки говорят о том, что один из вложенных процессов в настоящее время запущен (что является нормальной ситуацией), критические ошибки отмечаются красными крестиками. Вопросительные знаки указывают на то, что результаты данного процесса устарели и требуется его повторный запуск. При появлении ошибки трансляция прекращается. При успешном завершении трансляции в консоли выдается сообщение: «Process "Generate Programming File" completed successfully» (см. рисунок 2.39).

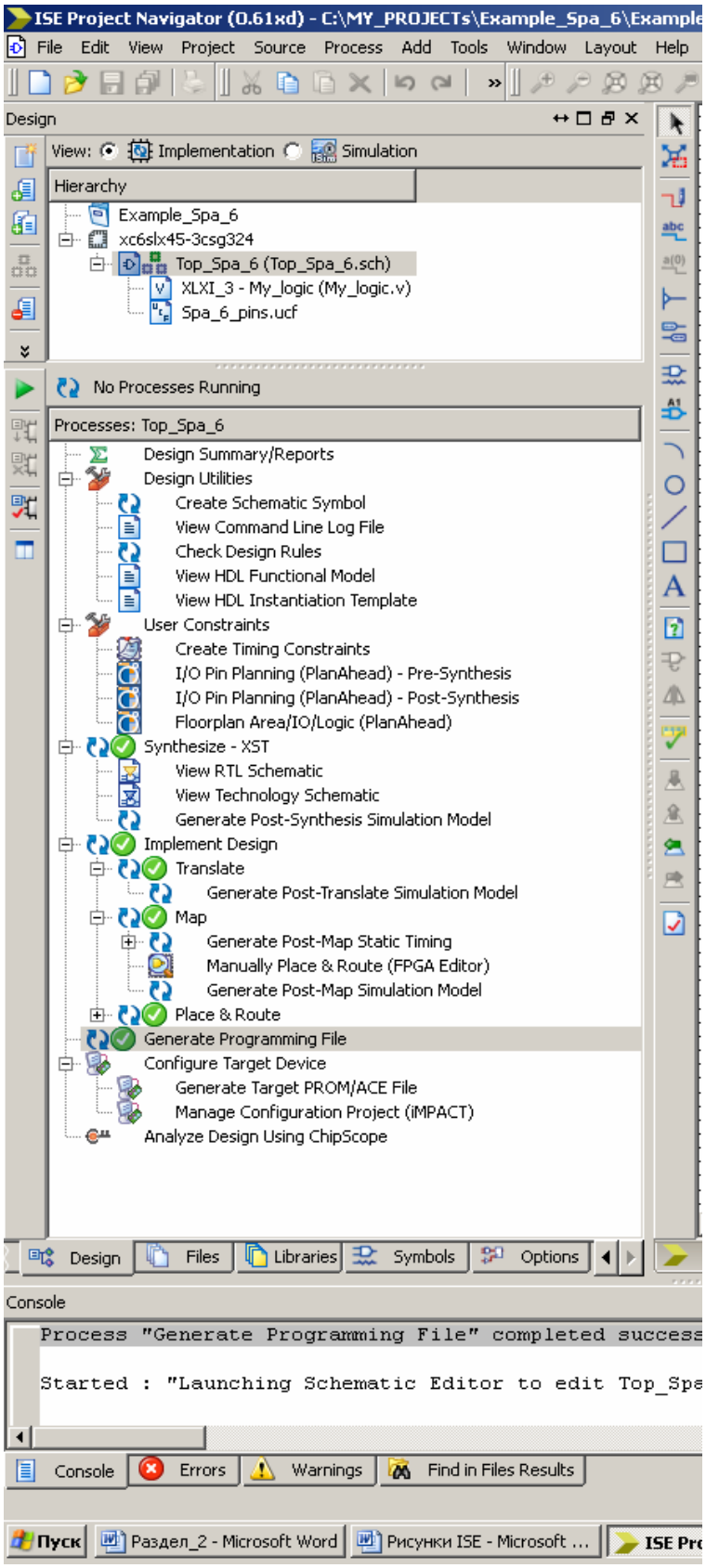


Рисунок 2.39 – Пример окна **Process** при успешном завершении процесса генерации битового файла прошивки ПЛИС.

После того, как процесс генерации файла прошивки успешно завершен, рекомендуется проверить, что в папке проекта создан соответствующий бинарный файл с расширением ***.bit*** (в данном случае – ***top_spa_6.bit***).

2.1.7 Временное моделирование и оптимизация проекта

Необходимо иметь в виду, что временные задержки, указанные оператором **#** в поведенческой модели, не являются указанием САПР обеспечить заданную задержку в проектируемом устройстве. Напротив, приведение этого значения означает, что разработчик каким-то образом спрогнозировал задержку распространения, определил ее из технической документации, либо планирует рассмотреть, каким бы было поведение устройства, если бы задержки были равны указанным. Реальные задержки зависят от используемой аппаратной платформы, размещения компонентов на кристалле и особенностей выполнения трассировки.

В САПР ПЛИС ISE возможно проведение моделирования на уровне, учитывающим физическую реализацию проекта, когда задержки распространения сигналов не принимаются равными тем, которые указаны в поведенческом описании, а рассчитываются, исходя из физических моделей компонентов ПЛИС и трассировки конкретного проекта. Для моделирования в таком режиме необходимо в выпадающем списке выбрать перед запуском моделирования один из возможных режимов **Post: Post-Translate,** или **Post-Map,** или **Post-Route** (см. рисунок 2.40).

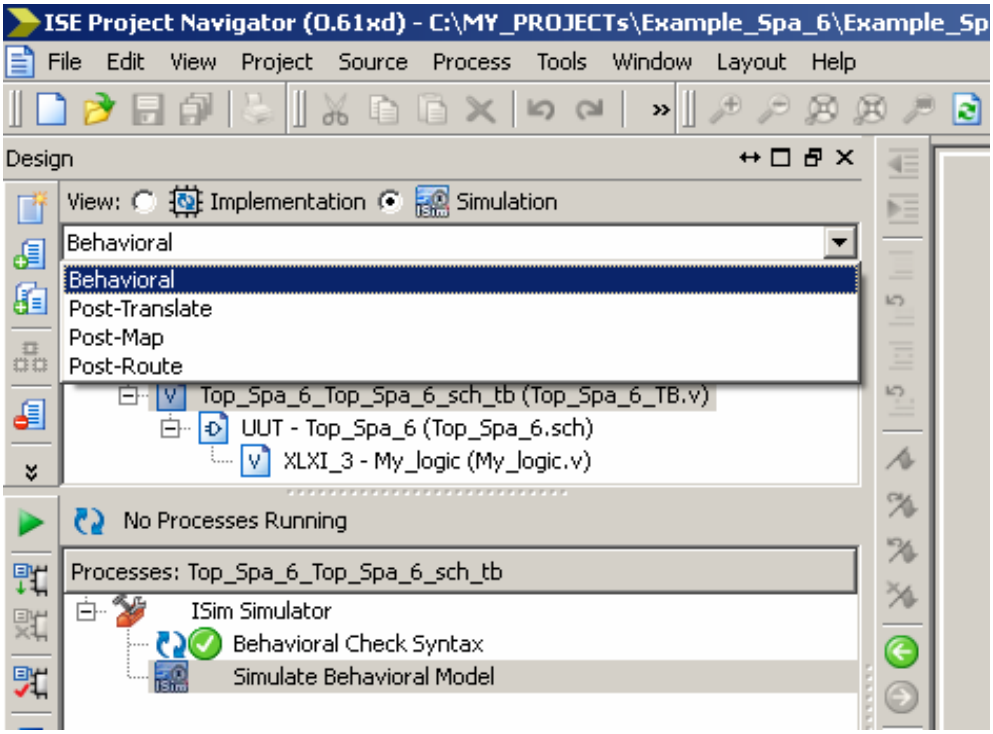


Рисунок 2.40 – Выпадающий список для выбора режимов моделирования.

В этом случае результаты моделирования компонента задержки сигналов будут определены с учетом влияния внутренних цепей и буферов, подключенных на соответствующем этапе к выводам ПЛИС.

Для получения результатов **Post-Route** моделирования необходимо, как следует из названия, выполнить трассировку (**Routing**) проекта. Таким образом осуществляется привязка абстрактного проекта к конкретной элементной базе. Выполнение моделирования в этом случае связано с необходимостью анализа физических моделей компонентов, что требует существенно большего времени по сравнению с поведенческим моделированием. Однако такой результат существенно точнее, поскольку времена распространения сигналов рассчитываются по реальной трассировке, а не вводятся в модель из субъективных соображений.

При проектировании цифровых систем моделирование на физическом уровне обычно используется на завершающих этапах верификации, когда требуется подтвердить не просто правильность выполнения преобразований, а соблюдение временных характеристик установления и распространения сигналов. В процессе отладки удобнее использовать моделирование на поведенческом уровне, поскольку оно производится существенно быстрее. В этом случае даже ориентировочные величины задержек позволяют иллюстрировать временной сдвиг сигналов друг относительно друга, что позволяет отлаживать архитектуру устройства.

Основным преимуществом моделируемого подмножества Verilog является возможность создания моделей, описывающих задержки распространения сигналов. Это позволяет применять такие задержки к элементам, основываясь только на информации, указанной разработчиком модели, что существенно быстрее, чем получение этой информации путем анализа физической модели этого компонента. Таким образом, разработчик модели обязан обеспечить правильные величины задержек, но это компенсируется увеличением скорости моделирования.

Задержка указывается с помощью символа **#.** Например, для непрерывного присваивания:

```
assign #3 q = a & b;
```

#3 показывает, что задержка распространения сигнала составляет 3 нс (точнее, 3 «единицы времени», величина которых определяется директивой `timescale, и обычно равна 1 нс).

Для логических вентилей могут указываться три величины задержек, соответствующие следующим величинам:

- время перехода в высокий уровень (rise time);
- время перехода в низкий уровень (fall time);
- время отключения (turn off time).

Эти времена указываются после символа # в скобках, в порядке, приведенном в списке.Например:

```
assign #(2,3,4) q = a & b;
```

Необходимо еще раз подчеркнуть, что приводимые таким образом задержки используются только при моделировании. Они игнорируются средствами синтеза, которые

вместо этого могут рассчитать реальные задержки распространения, учитывающие используемые компоненты, соединяющие их проводники, условия работы и т.д.

Другими параметрами, требующими анализа и оптимизации, являются мощность, потребляемая проектируемым устройством и ресурсы, задействованные для реализации данного устройства из состава стандартных компонентов, имеющихся в данном кристалле.

Инструменты анализа параметров по мощности сосредоточены в программном пакете **XPower** [2.1], а оптимизация ресурсов ПЛИС осуществляется на основе анализа таблиц, обобщающих результаты проекта, путем изменения настроек параметров синтеза. Подробное описание этих инструментов и процедур оптимизации синтеза приведено в книгах [2.2 и 2.4].

2.1.8 Программирование ПЛИС (загрузка проекта)

Заключительным шагом создания устройства на основе ПЛИС является операция конфигурирования интегральной схемы программируемой логики, т.е. программирование необходимых функций в базисе имеющихся в кристалле схемы ресурсов: элементов памяти, генераторов логических функций, триггеров, матриц коммутации цепей и.т.п. Для этого необходимо располагать специальными аппаратными средствами: программатором, с интерфейсом **JTAG** (о периферийном сканировании через цепи JTAG см. подробнее в [2.13]) для соединения с ПЛИС. Программатор должен быть подключен к компьютеру специальными кабелями через LPT-порт или через USB-порт, или через COM-порт.

В частности, на используемой в данном проекте демонстрационной плате ATLYS имеется встроенный программатор с интерфейсом USB. Для прошивки битового файла проекта на плату ATLYS требуется выполнить следующие действия:

1) Подсоединить плату к USB входу компьютера в соответствии со схемой соединений, представленной на рисунке 2.41.

2) Через специальный адаптер питания подключить плату к сети переменного тока 220В и включить питание платы: перевести движковый переключатель POWER, расположенный рядом с разъемом адаптера питания в положение ON.

3) Проверить, что устройство Digilent USB обнаружено и готово к работе.

4) В открытом навигаторе проектов ISE в окне **Process** запустить на исполнение процесс **Manage Configuration Project (iMPACT),** во вновь открывшемся окне приложения **ISE** – программы **iMPACT**, работающей в режиме периферийного сканирования (см. рисунок 2.42), вызвать функцию **Boundary Scan,** правой кнопкой мыши вызвать контекстное меню приглашения для запуска периферийного сканирования «***Right click to Add Device or Initialize JTAG chain»*** и нажать кнопку **Initialize chain**.

5) В результате в рабочем окне программы **iMPACT**, отражающем результаты сканирования, появится символическое изображение подключенных JTAG устройств, как это показано на рисунке 2.43. В появившемся диалоговом окне следует нажать кнопку **Yes**, т.е выбрать функцию назначения битового файла.

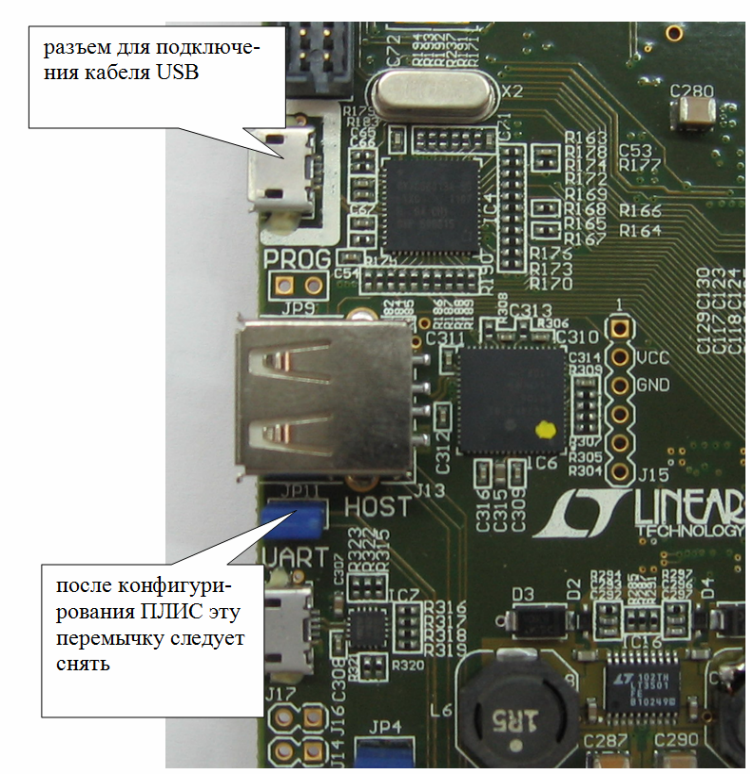


Рисунок 2.41 – Схема подключения демонстрационной платы к USB-порту для конфигурирования ПЛИС.

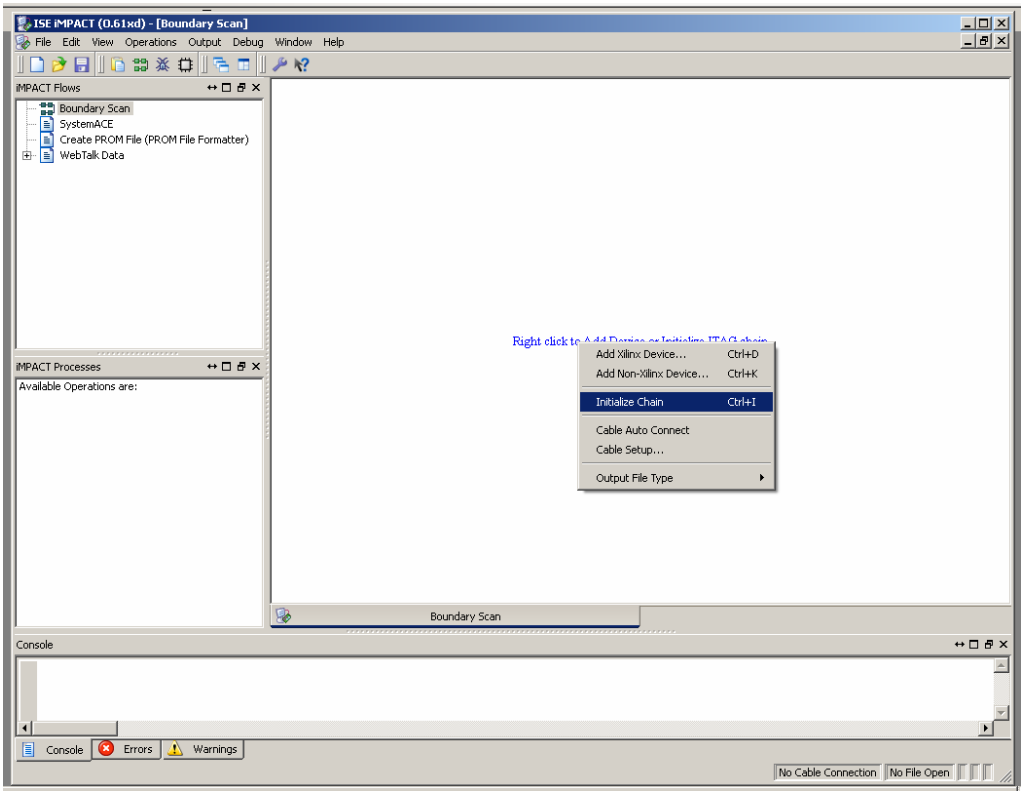


Рисунок 2.42 – Пример интерфейса программы **iMPACT** для программирования ПЛИС с активизированным окном периферийного сканирования микросхем.

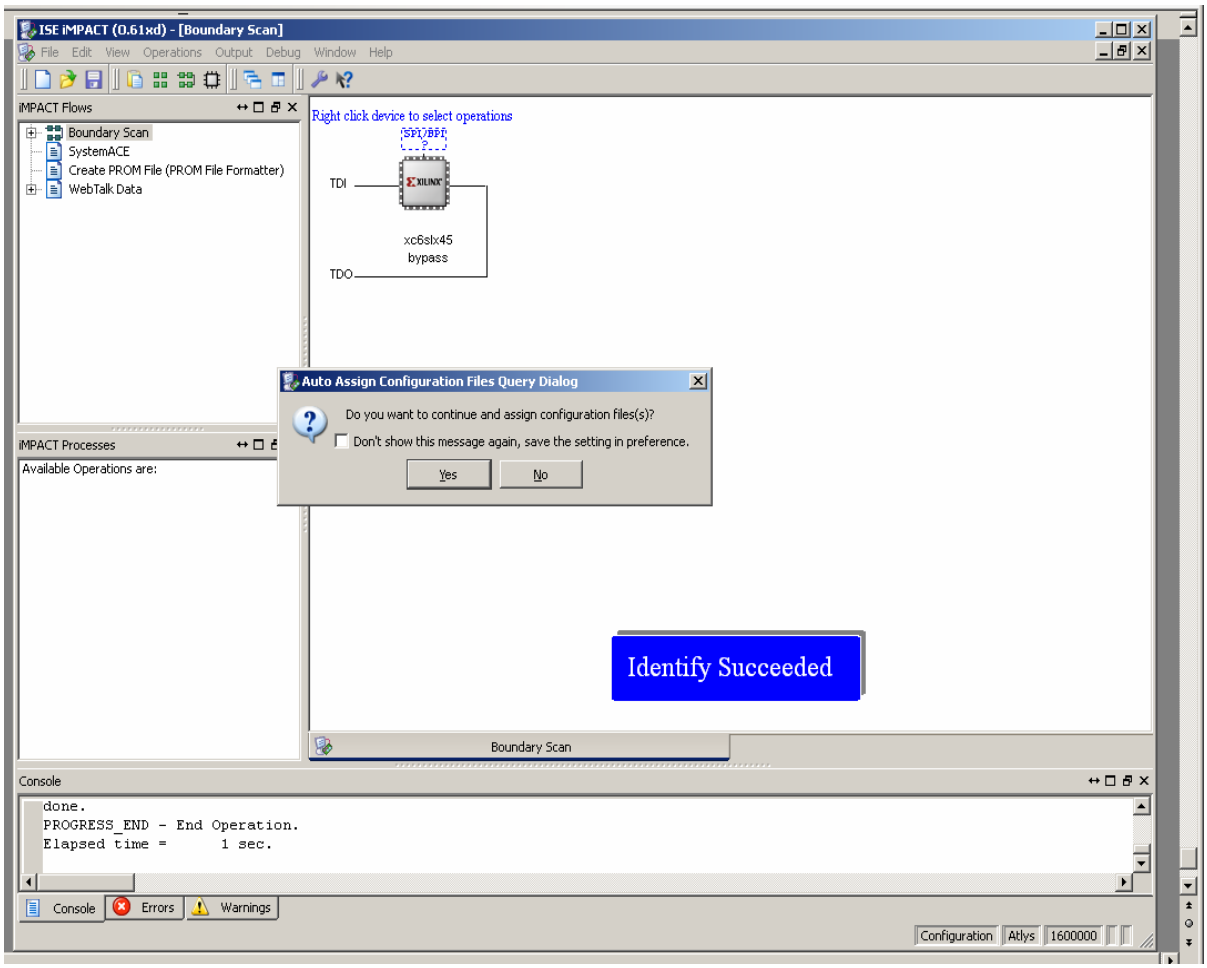


Рисунок 2.43 – Пример интерфейса программы **iMPACT** с результатами сканирования обнаруженных устройств (микросхема 6SLX45CSG324 на плате ALTYS).

6) В результате на экран будет выведено окно поиска требуемого файла, который должен иметь расширение **.bit** . Следует указать путь к файлу прошивки. Как показано на рисунке 2.44 для данного проекта – это файл ***top_spa_6.bit***, расположенный в папке ***C:\MY_PROJECTs\Example_Spa_6.***

7) После того, как файл будет выбран (кнопка **Open**) в окне сканирования JTAG устройств файла этот файл будет связан с соответствующей схемой ПЛИС, и появится диалоговое окно, предлагающее продолжить работу с встроенной в данную микросхему Flash-памятью. В данном случае этого не требуется. В результате появится диалоговое окно, представленное на рисунке 2.45, в котором файл прошивки связан с графическим изображением микросхемы ПЛИС.

В итоге для окончательной прошивки проекта в ПЛИС с помощью программы **iMPACT** следует активизировать изображение программируемой схемы в цепочке JTAG и в окне **Process** вызвать на исполнение процедуру **Program** (см. рисунок 2.44) и убедиться, что процесс программирования успешно завершен. О том, что микросхема запрограммирована, указывает также индикатор DONE расположенный на плате.

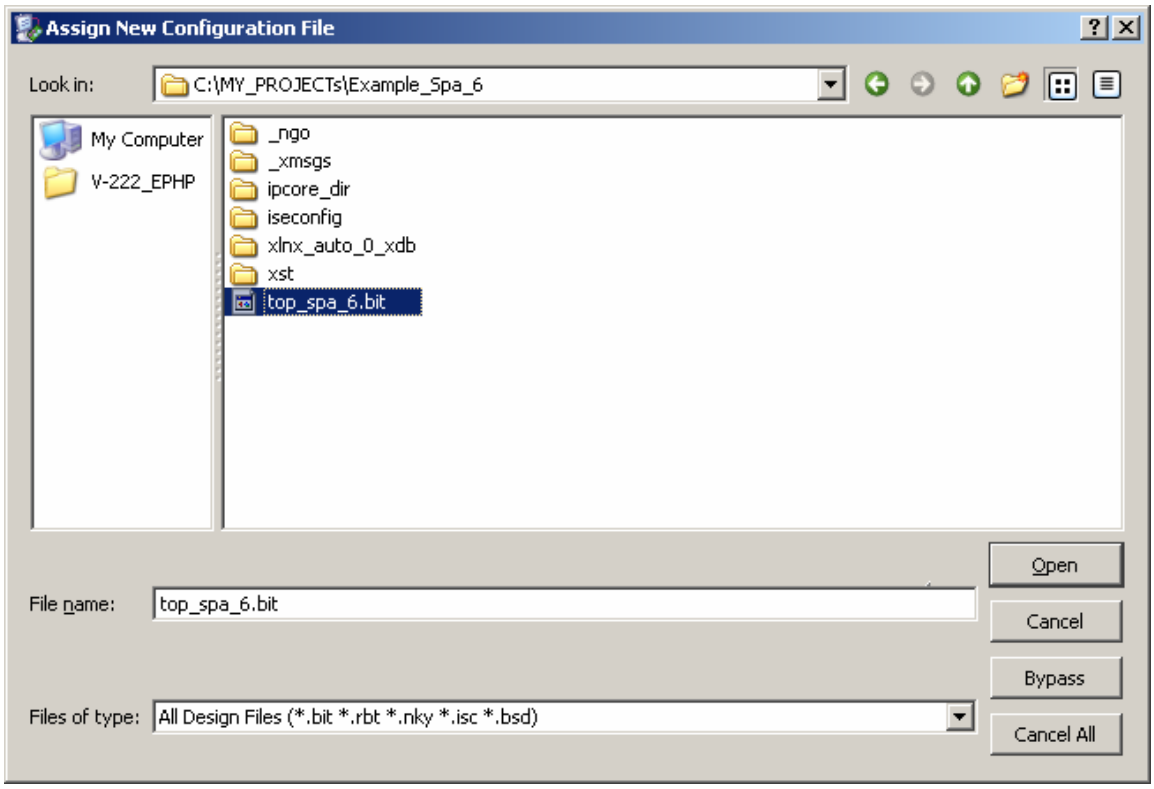


Рисунок 2.44 – Пример задания файла прошивки ПЛИС.

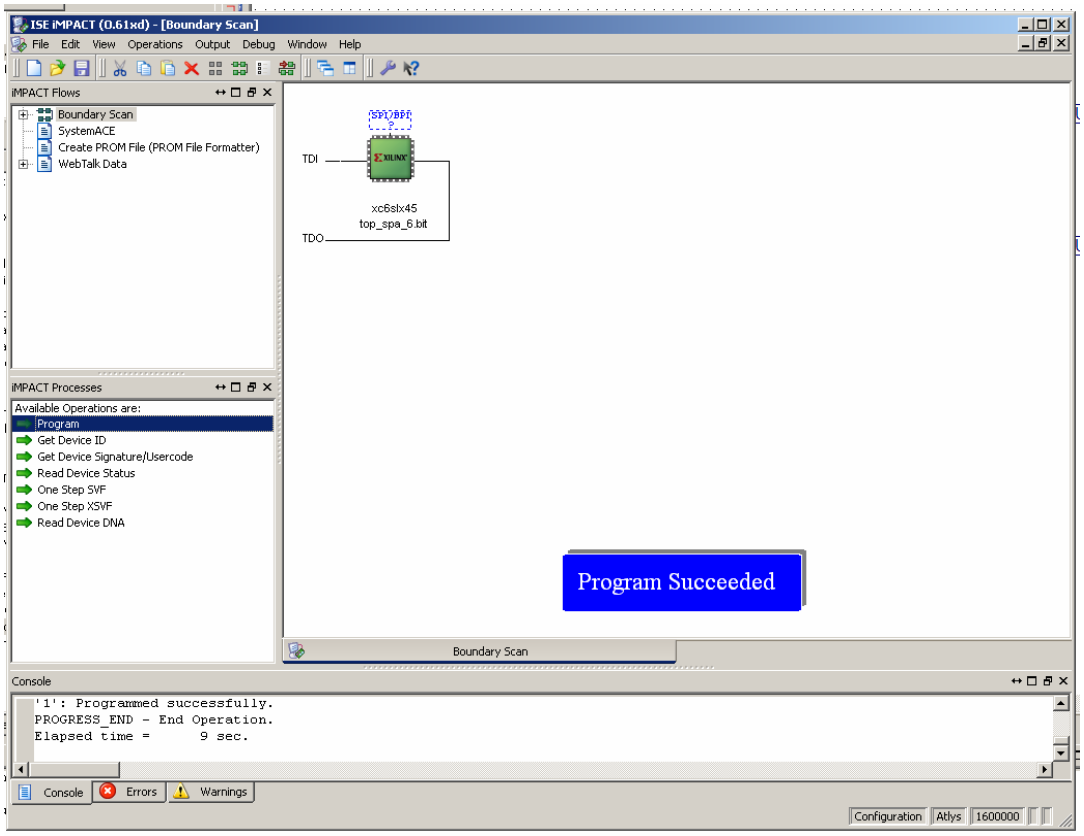


Рисунок 2.45 – Вызов процедуры программирования ПЛИС в программе **iMPACT**.

В итоге для окончательной прошивки проекта в ПЛИС с помощью программы **iMPACT** следует активизировать изображение программируемой схемы в цепочке JTAG и в окне **Process** вызвать на исполнение процедуру **Program** (см. рисунок 2.45) и убедиться, что процесс программирования успешно завершен. О том, что микросхема запрограммирована, указывает также индикатор DONE расположенный на плате.

Для переноса проекта устройства на другую систему с такими же подключаемыми сигналами и микросхемами (например, на другой идентичный экземпляр платы ATLYS), достаточно располагать бинарным файлом прошивки устройства (для данного проекта – это файл ***top_spa_6.bit***).

Заключительной стадией проектирования является тестирование (аппаратная верификация) проекта, т.е. проверка соответствия реальных результатов работы устройства ожидаемым.

В данном случае процесс тестирования заключается в том, что для заданных положений переключателей SW0 и SW1, определяющих сигналы на входах устройства следует по светодиодным индикаторам проверить значения выходных сигналов (положение «включено» соответствует логической единице, положение «выключено» – логическому нулю). Результаты аппаратного тестирования, приведены в таблице 2.6.

Таблица 2.6 – Результаты аппаратного тестирования разработанного устройства.

Комбинация положений переключателей		Ожидаемое состояние светодиодов		Наблюдаемое состояние светодиодов		Результат теста
SW0 (INPUT_A)	SW1 (INPUT_B)	LD0 (OUTPUT_1)	LD1 (OUTPUT_2)	LD0 (OUTPUT_1)	LD1 (OUTPUT_2)	+ или -
1	1	0	1	**0**	**1**	+
1	0	0	0	**0**	**0**	+
0	1	0	0	**0**	**0**	+
0	0	1	0	**1**	**0**	+

Таким образом, результаты тестирования показывают, что разработанное устройство работает правильно и проект можно считать успешно завершенным.

2.1.9 Подготовка технической документации проекта

Техническая документация проекта представляет собой комплект папок и файлов, которые созданы САПР ISE в папке проекта (в данном случае – в папке **Example_Spa_6**.

Результаты выполнения проектных процедур в среде САПР ISE сохраняются как текстовые документы – отчеты. Отчеты и обобщенная информация о выполняемом проекте доступны пользователю из вкладки **Design Summary** окна документов проектов (см. пример на рисунке 2.46). Чтобы открыть этот документ следует вызвать на ис-

полнение процесс **Design Summary/Report** из окна **Process**.

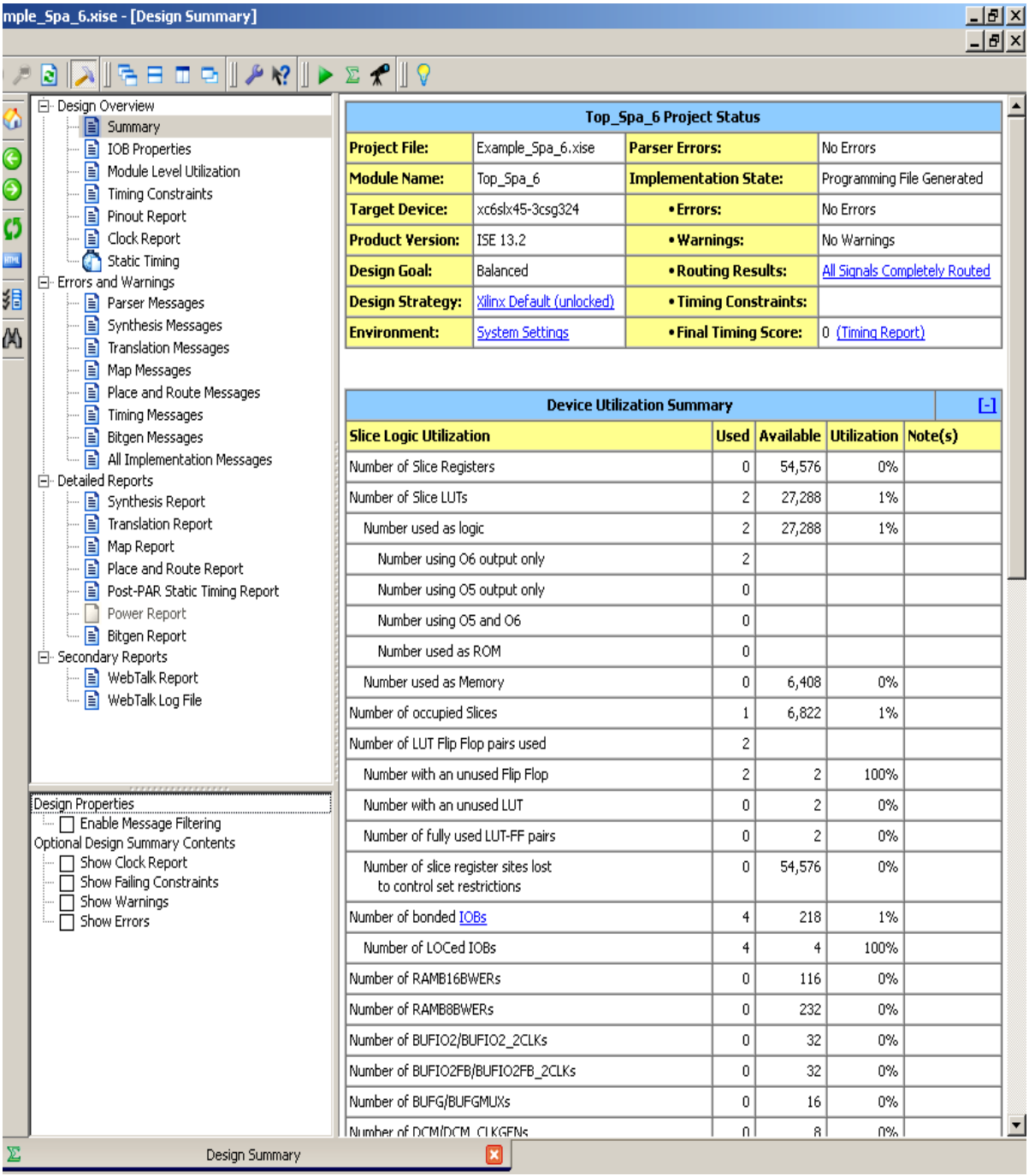


Рисунок 2.46 – Пример окна Design Summary с обобщенными данными о проекте.

Этот документ, в частности, включает следующие разделы: проводник отчетов проекта, таблица текущего состояния проекта (в данном случае – таблицу **Top_Spa_6 Project Status**), таблицу затраченных и свободных ресурсов ПЛИС (**Devices Utilization Summary**).

Пример информации, которая может быть извлечена их отчета о проекте: данные о временных задержках при реализации в ПЛИС приведен на рисунке 2.47

```
Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 4 / 1
-------------------------------------------------------------------------
Delay:               5.402ns (Levels of Logic = 3)
  Source:            INP_L_2<2> (PAD)
  Destination:       OUT_L_2 (PAD)

  Data Path: INP_L_2<2> to OUT_L_2
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   1.222   0.827  INP_L_2_2_IBUF
(INP_L_2_2_IBUF)
     LUT4:I0->O            1   0.203   0.579  Component_3/OUT_SUM1
(OUT_L_2_OBUF)
     OBUF:I->O                 2.571          OUT_L_2_OBUF (OUT_L_2)
    ----------------------------------------
    Total                      5.402ns (3.996ns logic, 1.406ns route)
                                       (74.0% logic, 26.0% route)

=========================================================================
```

Рисунок 2.47 – Файл отчета о временных задержках.

Литература к разделу 2:

2.1. http://www.xilinx.com
2.2. Зотов В.Ю. Проектирование цифровых устройств на основе ПЛИС фирмы XILINX в САПР WebPACK ISE. – М.: Горячая линия – Телеком, 2003. – 624 с.
2.3. Тарасов И.Е. Разработка цифровых устройств на основе ПЛИС XILINX® с применением языка VHDL. М.: Горячая линия - Телеком, 2005. – 252 с.
2.4. Тарасов И.Е., Потехин И.Е. Разработка систем цифровой обработки сигналов на базе ПЛИС. М.: Горячая линия - Телеком, 2007. – 248 с.
2.5. Амосов В.В. Схемотехника и средства проектирования цифровых устройств. БХВ-Петербург, 2007. – 542 с.
2.6. http://www. digilentic.com
2.7. http://www.mentor.com
2.8. http://www.cadence.com/us/pages/default.aspx
2.9. http://www.synopsys.com/home.aspx
2.10. Тарасов И. Е. Язык описания аппаратуры Verilog:Учебное пособие. М.: МГТУ МИРЭА, 2011. - 121 с.
2.11. Поляков А. К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. - М.: Изд-во "СОЛОН-Пресс", 2003. - 320 с
2.12. Стемпковский А.Л., Семенов М.Ю. Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: Учебное пособие. М.: МИЭТ, 2005. -140с.
2.13. http://www.jtag-technologies.ru

3 Язык описания аппаратуры Verilog

3.1 Общие сведения

С ростом сложности интегральных схем появилась необходимость во все более полном моделировании разрабатываемых микросхем перед их передачей в производство. Для решения этой задачи были разработаны *языки описания аппаратуры* (Hardware Description Languages, HDL). Их основным назначением было моделирование процессов, происходящих в цифровых микросхемах, на различных уровнях. При этом большое количество ошибок, проистекающих из-за некорректного планирования или допущенных конструктивных промахов, оказалось возможным идентифицировать уже на этапе моделирования, что существенно дешевле, чем исправлять ошибки путем повторного изготовления полупроводниковой пластины. Поскольку с уменьшением технологических норм стоимость технологической подготовки производства полупроводниковых микросхем существенно растет, их предварительное моделирование становится все более и более актуальным. Также эффективным является прототипирование цифровых устройств с помощью программируемых логических интегральных схем (ПЛИС).

Несмотря на то, что главным назначением HDL является моделирование цифровых устройств, в них было выделено *синтезируемое подмножество* – набор конструкций, которые могут быть преобразованы в список связей цифровых компонентов и далее в топологическое представление интегральной микросхемы или в конфигурационный поток для программирования ПЛИС. Кроме того, различные реализации HDL добавляют к стандартам специфические расширения, зависящие от конкретного производителя. Эти расширения в общем случае не являются переносимыми, поэтому на них следует обращать внимание при создании HDL-описаний, которые предполагается переносить в другие системы проектирования. Соотношение между подмножествами языков описания аппаратуры можно представить следующим рисунком:

Рисунок 3.1 - Соотношение между подмножествами конструкций в языках описания аппаратуры.

В настоящее время наиболее распространенными являются языки описания аппаратуры VHDL и Verilog HDL (Verilog) [3.1]. Они имеют одинаковые области применения, схожи по основным характеристикам, и отличаются в основном грамматикой. В целом можно сказать, что модули на Verilog являются более компактными, чем аналогичные модули на VHDL, однако это является следствием большей структурированности VHDL и более строгих требований к контролю типов и форматам описания основных синтезируемых конструкций. При этом алгоритмы синтеза схемотехнического представления цифровых систем являются настолько схожими по характеристикам, что не имеет практического смысла переходить с одного языка описания аппаратуры на другой, рассчитывая на улучшение параметров синтезированной схемы только вследствие более эффективной программы синтеза. В то же время, синтезаторы различных производителей могут давать различные результаты для относительно сложных цифровых узлов.

Стандарт на язык описания аппаратуры Verilog содержится в документе IEEE 1364. В настоящий момент основной для проектирования является версия стандарта от 2001 года (Verilog-01), однако предыдущая версия (Verilog-95) также может представлять интерес по соображениям с ранее выпущенными программными продуктами и описаниями цифровых систем. В 2005 году в стандарт были внесены несущественные изменения. Краткое описание основ языка можно найти в [3.1 и 3.2].

При разработке цифровых систем используется понятие *уровней абстрагирования*. Их смысл состоит в том, что при смене технологических процессов, полупроводниковых материалов, схемотехнических решений для элементарных цифровых узлов и прочих низкоуровневых деталей реализации было бы нежелательным терять ранее полученные результаты, касающиеся общей архитектуры системы и порядка соединения компонентов. Поэтому изменения в цифровые системы вносятся на различных уровнях, так что детали реализации конкретных цифровых узлов (например, триггера или сумматора) скрыты от разработчика общей структуры цифрового устройства. Впоследствии, если будет сконструирован более эффективный вариант триггера, все схемы, созданные с использованием такого компонента, могут быть использованы без необходимости их коренного пересмотра.

Обычно говорят о следующих уровнях абстрагирования:

1. Поведенческий (behavioral level)
2. Уровень регистровых передач (RTL, register transfer level)
3. Вентильный уровень (gate level)
4. Топологический уровень (cells level)
5. Физический уровень (switch/mask level)

Описание на **поведенческом уровне** является наиболее отдаленным от конкретной технической реализации и представляет цифровые модули в виде «черных ящиков» с заданной функциональностью. Языки описания аппаратуры предоставляют достаточно средств для работы на этом уровне, что обеспечивает высокую производительность труда разработчика цифровых устройств.

Уровень регистровых передач подразумевает представление цифрового устройства в виде схемы соединения регистров и комбинаторной логики.

На **вентильном уровне** производится детализация схемы до базовых логических элементов (вентилей). Таким образом, комбинаторные выражения, представляемые на предыдущем уровне в общем виде, получают развернутую форму, что позволяет определить требуемую сложность схемы, уточнить занимаемую ей площадь и задержки распространения сигналов.

На топологическом уровне цифровое устройство представляется в виде координат расположения отдельных фрагментов (ячеек), соответствующих имеющимся в технологической библиотеке компонентам.

Физический уровень содержит сведения о размещении и параметрах физических устройств (транзисторов), непосредственно реализующих необходимые функции.

Проектирование с использованием языков описания аппаратуры обычно производится с перекрытием поведенческого уровня и уровня регистровых передач. Это связано с тем, что конструкции, описываемые на поведенческом уровне, могут оказаться слишком неэффективными для используемой аппаратной базы именно в силу максимального абстрагирования (дистанцирования) от деталей реализации. Использование RTL-представления позволяет конкретизировать пожелания разработчика к способу реализации наиболее ответственных узлов и снизить степень влияния алгоритмов синтеза на получаемый результат.

Близко к RTL-представлению находится также *структурный подход* к описанию схем, при котором схема описывается с помощью указания цифровых элементов из конкретной библиотеки компонентов и схемы их соединения. Сравним следующие способы представления элемента 2И на Verilog.

```
out_c = in_a && in_b;
```

Данный текст транслируется программой синтеза с получением требуемого цифрового узла. Способ реализации и используемые для этого компоненты автоматически определяются средствами синтеза, на основе входной информации представляющей собой выражение на языке описания аппаратуры. Такой подход к получению компонентов в англоязычной литературе обозначается как ***inference***.

```
and2 and2_inst0 (.a(in_a), .b(in_b), .c(out_c));
```

Приведенное описание служит указанием на то, что требуется использовать некоторый модуль **and2**, имеющий входы **a** и **b**, а также выход **c**. Детали реализации такого модуля должны быть представлены в библиотеке, поставляемой фирмой-производителем САПР, а выводы конкретного экземпляра, названного **and2_inst0**, должны быть подключены к цепям **in_a, in_b, out_c** соответственно. Поскольку создается экземпляр («пример», *instance*) библиотечного компонента, подход обозначается как ***instantiation***.

Сравнение представленных подходов делает очевидным тот факт, что использование *instantiated*-компонентов сопряжено с более сложным форматом их описания и потенциальными проблемами отладки схемы, поскольку никакого анализа поведения и логической корректности подключения выводов не производится. Разработчик, используя ***instantiation***, должен представлять себе дальнейшее поведение создаваемой им схемы, и ответственен за правильное подключение внешних сигналов. Напротив, средства синтеза в процессе создания ***inferred***-компонентов способны выбрать корректное с точки зрения схемотехники решение, основываясь на достаточно абстрактном описании желаемого результата.

Развитие средств автоматического проектирования делает доступным автоматический синтез эффективных схем для все большего числа цифровых узлов. В настоящее время исключениями для ПЛИС, требующими использования ***instantiation***, являются в основном специфичные компоненты. Примерами сложат буферы специального назна-

чения, схемы формирования тактового сигнала, скоростные приемопередатчики и прочие компоненты, которые не могут быть представлены в виде HDL-описания. Кроме того, в ситуациях, когда различные настройки программ синтеза дают отличающиеся результаты, не устраивающие опытного разработчика, он имеет возможность ликвидировать эту неопределенность, непосредственно указав на библиотечные компоненты, с помощью которых он хотел бы реализовать требуемый фрагмент цифровой системы. Наконец, сборка цифровой системы на верхнем уровне иерархии также может быть произведена путем перечисления экземпляров компонентов и порядка их соединения. Такое описание является альтернативой графическому представлению схемы на верхнем уровне и в ряде случаев может оказаться более удобным, поскольку не зависит от конкретной версии редактора графического представления схем.

3.2 Основы языка Verilog

Основой описания на Verilog является *модуль*, представляемый в виде отдельного файла, обычно с расширением**.v**.

В САПР ISE подключение к проекту модулей на языке Verilog удобнее производить с помощью мастера создания новых компонентов (меню Project → New Source, или из контекстного меню области Design). В диалоговом окне мастера создания нового компонента необходимо выбрать тип файла «Verilog Module» и ввести имя в поле File Name, как показано на рисунке 3.2.

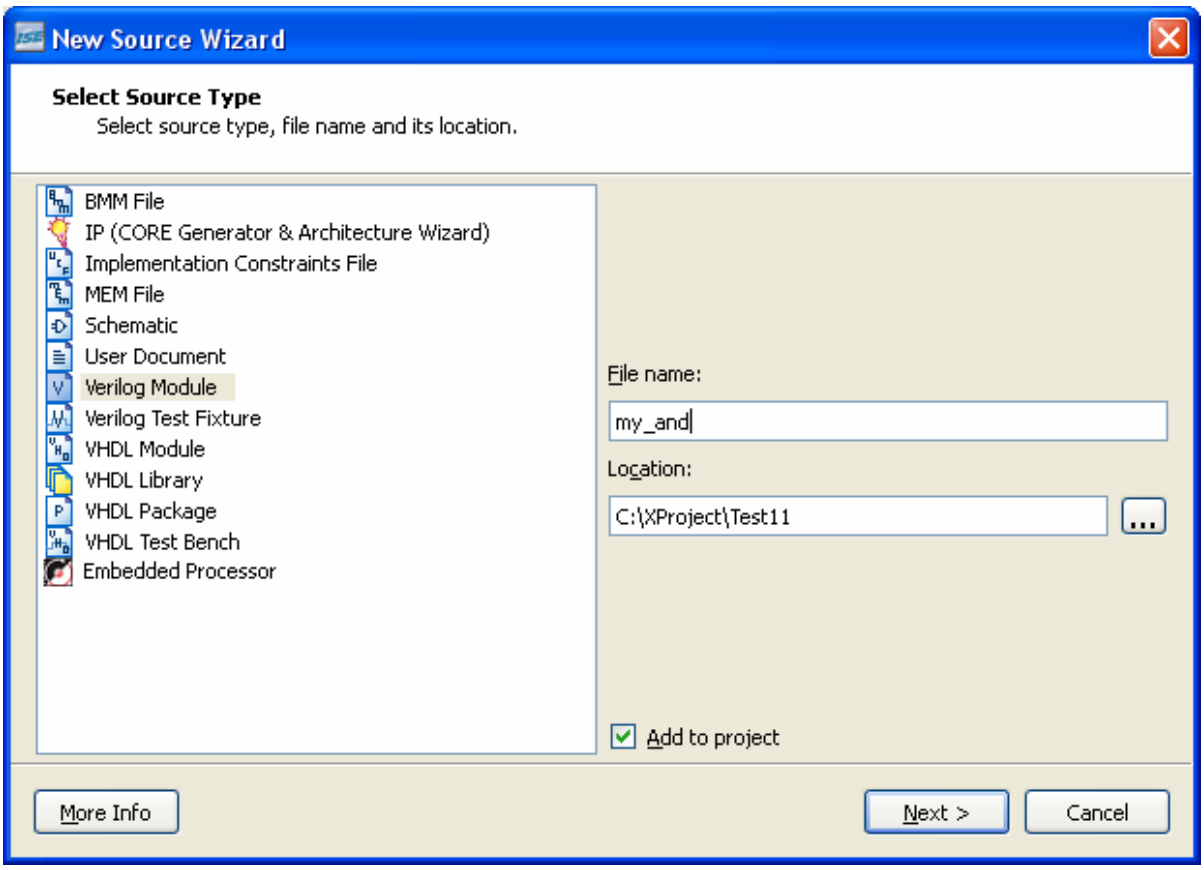


Рисунок 3.2 - Выбор модуля на языке Verilog в мастере создания нового компонента.

Структура модуля на языке Verilog представлена ниже.

```
module <имя> ([параметры] <объявления портов>)
[объявления локальных сигналов и переменных]
<синтезируемые конструкции>
endmodule
```

В качестве примера можно рассмотреть создание модуля, реализующего функцию логического И (см.рисунок 3.3)

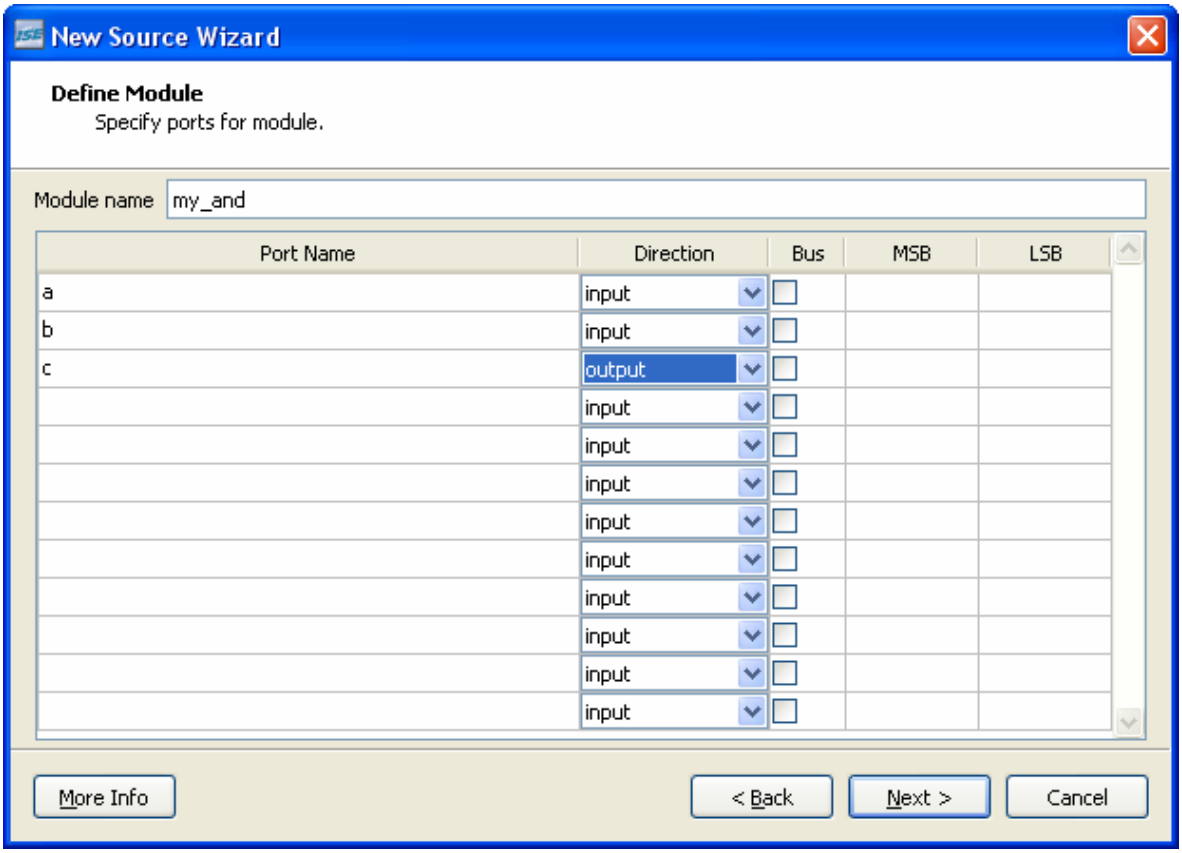


Рисунок 3.3 - Окно настройки параметров модуля в САПР ISE.

После завершения работы мастера будет сформирован шаблон с именем my_and.v, содержащий следующий текст (комментарии, добавленные в текст мастером, не показаны).

```
`timescale 1ns / 1ps
module my_and(
    input a,
    input b,
    output c
    );
endmodule
```

Директива `timescale предназначена для задания временных интервалов моделирования. *Первым символом `timescale является не апостроф, а знак, размещенный на клавише ~.*Первый из них, равный в приведенном примере 1 нс, задает длительность «единицы времени по умолчанию» для моделирования. В поведенческих описаниях, учитывающих время распространения сигналов, все временные интервалы с неуказанными единицами измерения будут рассматриваться как представленные в наносекундах. Второй параметр директивы `timescale задает величину дискретного шага, используемого средствами моделирования. В приведенном примере он равен 1 пс.

```
`timescale 1ns / 1ps
module my_and(
    input a,
    input b,
    output c
    );

assign #3 c = a & b;

endmodule
```

Представленный модуль содержит строку, реализующую его описание:

```
assign #3 c = a & b;
```

Оператор непрерывного присваивания `assign` предназначен для описания сигналов, формируемых с помощью комбинаторной логики. В данном случае выражение `c = a & b` синтезирует схему 2И с входами a и b и выходом c. Фрагмент описания #3 говорит о том, что установление нового значения происходит через 3 нс («3 единицы времени») после изменения сигналов на входах.

Полученное описание добавляется в проект немедленно и его поведение может быть промоделировано как на поведенческом (путем анализа текста на Verilog), так и на физическом (путем анализа физических моделей отдельных компонентов для конкретной аппаратной реализации) уровнях.

3.3 Типы данных

В Verilog существуют два основных типа данных (т.е. сигналов и их состояний): цепи **(net)** и переменные **(variable)**. Ключевое различие между ними состоит в поведении при моделировании.

Цепи постоянно отслеживаются, и их состояние обновляется немедленно после изменения состояния сигналов, от которых зависит состояние цепи.

Состояние **переменных** обновляется только внутри специальных *процедурных блоков*, специально вносимых разработчиком в текст модуля на Verilog.

Существуют следующие **разновидности цепей**:

- **wire** – обычное соединение между двумя узлами электрической схемы;
- **tri** – трехстабильное соединение;
- **wand, triand** – монтажное И;

- **wor, trior** – монтажное ИЛИ;
- **tri0** – притягивающий резистор к уровню земли;
- **tri1** – притягивающий резистор к уровню питания;
- **trireg** – цепь, имеющая емкость;
- **supply0** – уровень земли;
- **supply1** – уровень питания.

Поскольку Verilog предназначен в первую очередь для моделирования, в том числе и цифровых систем, состоящих из нескольких микросхем, соединяемых печатными проводниками, большинство цепей из представленного списка не имеют синтезируемого представления, а предназначены для моделирования соединений. Например, «монтажное И», «монтажное ИЛИ» применимы при объединении сигналов резисторно-транзисторной (РТЛ) или транзисторно-транзисторной логики (ТТЛ) на уровне печатной платы.

Основным типом цепей при проектировании цифровых систем на базе ПЛИС является **wire**.

Может также применяться **tri,** для случаев, относящихся к аппаратно реализованным буферам с третьим состоянием. Начиная с технологических норм 90 нм, основные производители ПЛИС с архитектурой FPGA отказались от реализации аппаратных буферов с третьим состоянием для внутренних цепей. Использование буферов с тремя состояниями ограничивается выходными линиями FPGA.

Сигналы с третьим состоянием могут быть объявлены для внутренних цепей ПЛИС и объединены с использованием схем разрешения одного из выходов, однако средства синтеза заменяют такие описания на эквивалентные узлы мультиплексируемой комбинаторной логики.

Основным типом **переменных** в Verilog является **reg**.

Это объект, который сохраняет записанное в него значение, и изменяет его только в явно определенные моменты. Переменная типа **reg** не означает автоматического применения триггера для ее реализации. Соответствующая синтезируемая схема может быть выполнена с использованием синхронной либо комбинаторной логики в зависимости от того, как именно описано присваивание новых значений такой переменной.

Сигналы внутри процедурных блоков **`initial`** и **`always`** могут присваиваться только объектам, имеющим тип **`reg`**. Например, мультиплексор на Verilog может быть описан одним из следующих способов.

1) С помощью комбинаторной логики и оператора непрерывного присваивания **`assign`**:

```
module MUX2 (input  A, B, SEL, output wire OUT1);
  assign OUT1  = ( A & SEL ) | ( B & ~SEL );
endmodule
```

2) С помощью процедурного блока `always`:

```
module MUX2 (input  A, B, SEL, output reg OUT1);
always @ (A, B, SEL)
     if (SEL)   OUT1 = A;
     else       OUT1 = B;
endmodule
```

В приведенных выше описаниях разные стили обуславливают разный тип выхода OUT1. Во втором случае, поскольку внутри модуля присваивание нового значения осуществляется внутри процедурного блока **always**, выход OUT1 объявлен как **reg**. В первом же варианте мультиплексора он имеет тип **wire** (это ключевое слово может быть опущено, поскольку wire является типом по умолчанию).

Синхронная (тактируемая) логика может быть описана только процедурным блоком:

```
module DFF (input CLK, D, output reg Q) ;
always @ (posedge CLK)
   Q = D;
endmodule
```

Назначение новой величины выходу **Q** производится только внутри процедурного блока, и только в моменты, когда выполняется условие **posedge CLK** (т.е. переход уровня сигнала **CLK** от логического нуля к логической единице). Поэтому для такого описания синтезируется D-триггер (D flip-flop).

3.4 Форматы представления значений

Основными значениями, которые могут быть присвоены цепям и переменным, являются:

0 – логический ноль;

1 – логическая единица;

z – «третье состояние», высокий импеданс;

x – неизвестное состояние (unknown).

Неизвестное состояние не имеет синтезируемого аналога, а используется при моделировании. Оно может возникнуть, когда в процессе моделирования выявился электрический конфликт. Например: два источника сигнала попытались задать различные логические уровни для одной и той же цепи.

Неизвестное состояние может быть искусственно сформировано разработчиком, если он планирует таким образом выявить возникновение ситуации, которая допустима с точки зрения электроники, но неприемлема по соображениям нормальной работы устройства. В этом случае по наличию состояний «x» можно будет быстро идентифицировать блоки, работающие некорректно.

Для задания чисел используется следующий формат:

[разрядность]'[s][основание]<значение>

где «разрядность» - разрядность числа в битах, представленная в десятичной системе счисления;

‘ – символ «апостроф»

s – необязательный модификатор, указывающий на то, что число представлено в формате со знаком

«основание» – модификатор основания системы счисления:

b – двоичная система счисления;

o – восьмеричная система счисления;

d – десятичная система счисления (по умолчанию);

h – шестнадцатеричная система счисления.

«значение» - значение числа, записанное в системе счисления, установленной модификатором.

Примеры:

8’b11110000 – двоичное 8-разрядное число 11110000 (240_{10});
16’hffff – шестнадцатиричное 16-разрядное число ffff (65535_{10});
8’d123 – десятичное 8-разрядное число 123.

Явное указание разрядности позволяет правильно выбрать количество незначащих разрядов, заполняемых нулями (для беззнакового представления чисел), или позицию знакового разряда (для представления чисел со знаком). Например, десятичное число 123 формально может быть записано с использованием только 7 разрядов, поэтому модификатор 8’ показывает, что в действительности следует использовать дополнительный, 8-й, разряд, поместив в него 0.

Если указанная в модификаторе разрядность числа больше, чем количество записанных разрядов, число автоматически расширяется до заданной разрядности:

6’b1111 → 001111

Если указанная в модификаторе разрядность числа меньше, чем количество записанных разрядов, число автоматически усекается.

4b’11000011 → 0011

Если разрядность числа не указана, она принимается равной 32. Однако при синтезе разрядность определяется исходя из разрядности переменной или цепи, которой оно присваивается.

В записи числа можно использовать символы подчеркивания, которые игнорируются при анализе числа и служат для повышения читаемости текста.

Поскольку объявление **`reg a`** соответствует одноразрядному значению, для хранения многоразрядных чисел переменные и цепи объединяются в *шины* (bus). Формат объявления многоразрядной переменной следующий:

```
reg [7:0] data = 255;
```

В этом примере объявляется 8-разрядная переменная `data`, разряды которой нумеруются от 7 до 0. При создании переменной присваивается значение 255.

Числа в скобках соответствуют *индексу* крайнего левого и крайнего правого разрядов в представлении числа. В представленном виде они совпадают со старшим значащим разрядом (**MSB, Most Significant Bit**) и младшим значащим разрядом (LSB, **Least Significant Bit**), однако допустима и форма записи, при котором первым указывается наименьший индекс:

```
reg [0:7] data = 8’b10000000;
```

В результате, исходя из формата объявления, присваивание константы 10000000_2 будет производиться так, что единица в старшем (крайнем левом) разряде записи константы будет присвоена 0-му разряду (крайнему левому в объявлении индексов) переменной `data`, и т.д. Таким образом, хотя двоичная константа может быть прочитана как 128_{10}, при ее записи в переменную действительным значением окажется 1. Поскольку для человека привычнее форма записи, при которой старший разряд находится слева, удобнее использовать формат объявления шин, в котором индексы указаны по убыванию , т.е. [7:0] а не [0:7].

Тип данных **integer** также является распространенной формой объявления переменных:

```
integer <name> [= <value>];
```

Пример:

```
integer data = 255;
```

Целочисленные переменные по умолчанию имеют разрядность 32.

В версии стандарта Verilog-95 переменные типа **integer** не могут инициализироваться в объявлении, для этого требуется отдельный оператор в процедурном блоке:

```
integer data;
initial
    data = 255;
```

Следующие типы данных не являются синтезируемыми и служат для повышения информативности моделирования.

Тип **real** представляет числа с плавающей точкой:

```
real x = 1.23;
real y = 3.00e8;
```

Применение чисел такого типа ограничивается в основном вычислением задержек распространения сигналов, а также выполнения математических вычислений, на основании которых можно принять решение о генерации того или иного вида синтезируемого кода. Например, в качестве параметра в модуль на Verilog может быть передана емкость нагрузки или температура окружающей среды, которые удобно представить в формате с плавающей точкой. На основании анализа этих величин могут быть автоматически сгенерированы различные варианты синтезируемых узлов.

Тип **time** предназначен для задания временных задержек:

```
time Tpd = 1 ns;
```

Допустимы следующие единицы измерения:

– fs – фемтосекунда;
– ps – пикосекунда;
– ns – наносекунда;
– us – микросекунда;
– ms – миллисекунда;
– s – секунда.

Можно еще раз отметить, что объект типа **time** не является синтезируемым, т.е. невозможно создать синтезируемую конструкцию с требуемой задержкой, просто указав ее величину в операторе присваивания. Подобные типы переменных присутствуют в Verilog для облегчения создания возможно более адекватных моделей проектируемых устройств. Таким образом, разработчик, применяя тип time, самостоятельно отвечает за корректность приводимых им величин задержек.

Тип **string** используется для хранения строк, используемых при выводе текстовых сообщений в процессе моделирования или работы алгоритмов синтеза:

```
Info_string = "This is a message from Verilog module"
```

Это также несинтезируемый тип, коды символов не формируют регистров или иных запоминающих устройств, а используются только для удобного задания текстовой информации, которую разработчик желал бы показать в консоли САПР или в генерируемом отчете.

3.5 Массивы

Такие устройства, как память, удобно рассматривать в виде ***массивов***. Verilog предоставляет возможность описывать массивы с помощью конструкции вида:

```
reg [7:0] memory [0:1]
```

В данном случае `memory` представляет собой не зарезервированное слово, а имя идентификатора, назначенного разработчиком. Объявляется массив 8-разрядных ячеек, к которым можно обращаться как `memory[0], memory[1]`.

Допустимо объявление многомерных массивов. Например:

```
reg [7:0] memory [0:7] [0:7]
```

Обращение к отдельным ячейкам производится указанием двух индексов:

```
memory [1][5].
```

Следует помнить, что технически многомерные массивы представляют собой линейную последовательность ячеек, а несколько индексов, задающих «координаты» ячейки, используются лишь для удобства иллюстрирования обращения к многомерным структурам данных.

Таким образом технически можно считать, что 8-разрядная ячейка данных является в действительности одномерным массивом отдельных разрядов. Принципиальным отличием от массивов является то, что для разрядов допустимы индивидуальные операции:

```
a[7] = b[4]
```

В этом примере 7-му разряду переменной a присваивается значение 4-го разряда переменной b.

Возможны операции с группами разрядов, однако количество разрядов в группах с левой и правой стороны от знака присваивания должно совпадать:

```
a[7:5] = b[3:1]
```

В этом примере указаны по 3 разряда обеих переменных.

Стандарт Verilog-2001 допускает еще одну форму задания разрядов, выбираемых из многоразрядной шины. Например, если объявлена шина a [31:0], то для выбора ее части можно написать:

```
a [31-:8] – аналогично записи a [31: 24]
a[16+:8] – аналогично записи a [23:16]
```

Можно увидеть, что первое число задает стартовый разряд, за которым идет символ + или -, означающие соответственно увеличение или уменьшение индекса. После двоеточия указывается не конечный разряд, а величина приращения/уменьшения индекса. Такая форма позволяет более наглядно показать в исходном тексте на Verilog ширину того фрагмента шины, который выбирается.

3.6 Порты

Модули, описанные на языке Verilog, обязательно имеют объявление *портов* – внешних выводов «черного ящика», поведение которого описано в теле этого модуля.

Структура модуля на языке Verilog.

```
module <имя> ([параметры] <объявления портов>)
[объявления локальных сигналов и переменных]
<синтезируемые конструкции>
endmodule
```

Существуют три типа портов:

- вход (input);
- выход (output);
- двунаправленный вывод (inout).

Входы и двунаправленные выводы всегда должны иметь тип **`wire`**. Это связано с тем, что значение для входов должно быть известно в любой момент времени, что обеспечивается именно типом `wire`.

Для выходов же возможна ситуация, когда их значение не будет изменяться при данном наборе входных сигналов. В этом случае выход будет удерживать состояние, присвоенное ему ранее, что возможно при использовании сигнала типа `reg`. В этом случае выходами служат сигналы типа **`reg`**.

Все сигналы *по умолчанию* имеют тип `wire`. При необходимости изменить тип на `reg` каждый сигнал должен быть упомянут отдельно:

```
module reg_sum (input [7:0] a, b,
                input clk, en,
                output reg [7:0] sum
                                  );
….
          always @ (posedge clk)
               if (en == 1) sum = a + b;
    endmodule
```

В приведенном примере явно указано, что выход имеет тип `reg`. Это позволяет присваивать выходу новое значение суммы не по каждому фронту сигнала `clk`, а только в те моменты, когда присутствует сигнал `en` («разрешение работы»). В противном случае значение выхода не изменяется. Попытка использовать в этом случае сигнал типа `wire` приведет к тому, что возникнет неопределенная ситуация – в моменты време-

ни, когда сигнал разрешения работы отсутствует, значение выхода не сможет быть назначено равным сумме входных сигналов. В то же время wire не дает возможности использовать элемент памяти для хранения предыдущего состояния, поэтому для того, чтобы в схеме появился регистр, для выхода sum использован тип reg.

Стандарт Verilog-95, который, возможно, мог бы потребоваться в рамках совместимости с программным обеспечением ранних версий, требует двукратного объявления сигналов типа reg.

```
module reg_sum (input [7:0] a, b,
                input clk, en,
                output [7:0] sum
                reg [7:0] sum
                              );
```

В первой из выделенных строк объявляется сигнал `sum`. Во второй указывается, что он имеет тип `reg`, при этом должна быть повторена спецификация размера. Такой способ, очевидно, менее удобен, и содержит избыточную информацию, и стандарт 2001 года допускает более компактное объявление сигналов.

3.7 Соединение модулей

Для создания проектов из нескольких модулей требуется их объединение. Причины разбиения проекта на модули могут быть различны, начиная от простого ограничения сложности каждого отдельного модуля, и заканчивая необходимостью использовать аппаратные узлы, имеющиеся в составе ПЛИС, которые включаются в проект с помощью библиотечного представления. Предположим, что необходимо создать схему, аналогичную показанной на рисунке 3.4

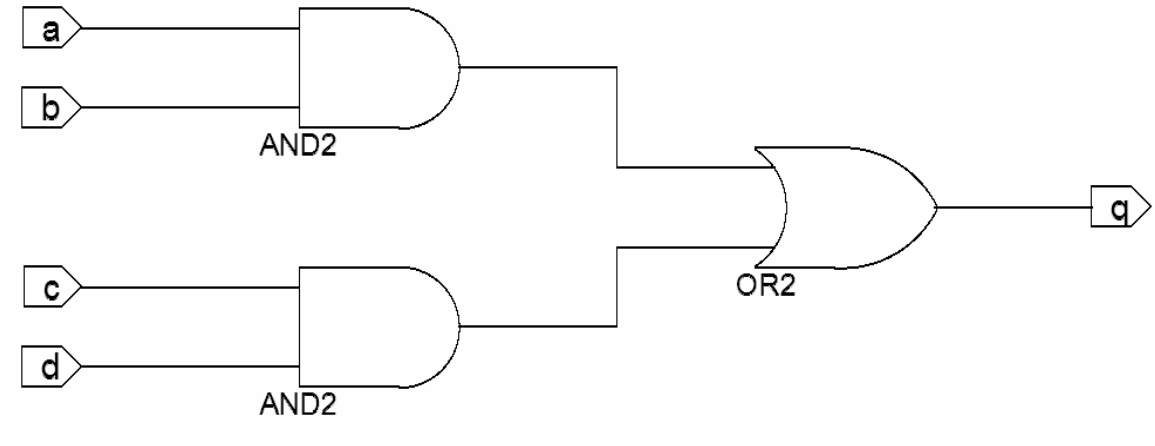


Рисунок 3.4 - Графическое изображение схемы из нескольких модулей.

Требуется, чтобы в библиотеке компонентов имелись модули, соответствующие компонентам `and2` и `or2`. Их описания могут быть следующими:

Модуль `and2`:

```
module and2(input a,
 input b,
output q);

assign c = a & b;

endmodule
```

Модуль `or2`:

```
module or2(input a,
           input b,
           output q);

assign c = a | b;

endmodule
```

На языке Verilog модуль верхнего уровня иерархии этого устройства будет иметь следующий интерфейс:

```
module top(input a,
           input b,
           input c,
           input d,
           output q);
```

Внутри модуля верхнего уровня должны присутствовать *два компонента* `and2` и *один компонент* `or2`. Они описываются с применением подхода *component instantiation*: ранее созданные компоненты указываются в тексте модуля верхнего уровня с перечислением сигналов, которые необходимо подключить к их портам. Однако при взгляде на схему можно заметить, что для соединения выходов компонентов `and2` с входами `or2` требуются два проводника, которые являются внутренними цепями и не имеют отдельного имени. Стандарт Verilog ***не допускает прямое соединения выхода одного компонента с входом другого*** (т.е., например, нельзя указать для порта a компонента `or2`, что он подключен непосредственно к порту `q` компонента `and2`). Для этого требуется ***объявление отдельных внутренних цепей***:

```
wire net1, net2;
```

Схема соединений внутри модуля `module top` описывается следующей конст-

рукцией:

```
        and2 component1(.a(a), .b(b), .q(net1));
        and2 component2(.a(c), .b(d), .q(net2));
        or2 component3(.a(net1), .b(net2), .q(q));
endmodule;
```

Цепи порта верхнего уровня, выделены в данном примере жирным шрифтом, чтобы отличать их от совпадающих имен портов компонентов (такое совпадение не является ошибкой).

Рассмотрим элементы записи:

and2 component2 (.a(c),

здесь:

and2 – имя компонента, как оно приведено в его описании после ключевого слова module.

component2 - уникальный идентификатор, присвоенный разработчиком ***данному экземпляру*** компонента (в схеме присутствует также component1 такого же типа).

.a – ссылка на порт a компонента and2;

c – указание цепи модуля верхнего уровня module top, к которой должен быть подключен этот порт.

Имена портов компонента можно не указывать, описывая только список сигналов, к которым необходимо подключать порты:

```
and2 component1(a, b, net1);
and2 component2(c, d, net2);
or2 component3(net1, net2, q);
```

В этом примере подключение к портам компонента осуществляется в порядке их описания в самом компоненте. Это так называемое *позиционное назначение* сигналов *(ordered mapping)* когда позиция сигнала в списке подключений совпадает с позицией порта в описании компонента. Предыдущий вариант, с явным упоминанием имен портов компонента, называется *именованным назначением (named mapping)*.

Нетрудно видеть, что позиционное назначение несколько компактнее, поскольку не включает в себя имена портов компонента. Тем не менее, именованное назначение позволяет избежать неявных ошибок, когда разработчик перепутал порядок следования имен. В качестве примера рассмотрим абстрактный компонент, который имеет сигналы сброса и разрешения работы:

```
module example(input clk, // тактовый сигнал
               input data, // сигнал данных
               input reset, // сигнал сброса
    input en, // сигнал разрешения работы
```

```
...
<прочие сигналы>
```

Не вдаваясь в подробности реализации такого модуля, рассмотрим варианты его подключения с использованием позиционного назначения сигналов:

```
example inst0(net1, net2, net3, net4);
example inst1(clk_net, data_net, en_net, reset_net);
example inst2(clk_net, data_net, reset_net, en_net);
```

Первый вариант объявления не дает возможности быстро убедиться, что подключение выполнено правильно, поскольку имена цепей отличаются только номером.

Второй вариант (`inst1`), использует «содержательные» имена, которые включают в себя фрагменты имен портов – например, `clk_net` дает понятие о том, что этот сигнал следует подключить к порту `clk`. Однако требуется отдельно сопоставить порядок объявления портов в описании модуля с порядком упоминания сигналов, чтобы убедиться, что для `inst1` перепутан порядок следования сигналов `en_net` и `reset_net`. Позиционное назначение не позволит САПР выявить или исправить эту логическую ошибку, поскольку никаких формальных запретов на подключение сигнала `en_net` к входу `reset` не существует.

Сравним это с именованным назначением:

```
example inst3(.clk(clk_net), .data(data_net),

.en(en_net),.reset(reset_net));
```

В этом примере порядок упоминания сигналов `en` и `reset` также не совпадает с порядком объявления сигналов в описании модуля. Однако явное упоминание имен сигналов заставит САПР подключить цепь `en_net` именно к входу `en`, несмотря на то, что на третьем месте в объявлении компонента находится `reset`. Таким образом, именованное назначение представляется более безопасным с точки зрения устранения логических ошибок при соединении компонентов.

3.8 Операторы языка Verilog

Язык Verilog предлагает разнообразные операторы, которые принято разбивать на группы.

3.8.1 Побитные операторы (bitwise)

Данные операторы предназначены для описания логических операций над сигналами и шинами, которые выполняются над соответствующими парами разрядов (битов). Побитные операторы представлены в таблице 3.1.

Таблица 3.1 - Побитные операторы языка Verilog.

Оператор	Название	Пример
&	И (AND)	`c = a & b;`
~&	И-НЕ (NAND)	`c = a ~& b;`
|	ИЛИ (OR)	`c = a | b;`
~|	ИЛИ-НЕ (NOR)	`c = a ~| b;`
^	ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)	`c = a ^ b;`
~^	ИСКЛЮЧАЩЕЕ ИЛИ-НЕ (NXOR)	`c = a ~^ b;`
~	НЕ (NOT)	`c = ~a;`

Таблицы истинности для побитных операторов сведены в таблицу 3.2.

Таблица 3.2 - Таблицы истинности для побитных операций.

A	B	И	ИЛИ	ИСКЛЮЧАЮЩЕЕ ИЛИ
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

В таблице не приведены варианты для инверсных операторов – И-НЕ, ИЛИ-НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ. Их значения противоположны (инверсны) результатам выполнения «прямых» операций, т.е. получаются заменой значений 0 на 1 и наоборот.

3.8.2 Арифметические операторы (arithmetic)

Арифметические операторы реализуют основные арифметические действия над сигналами и шинами (см.таблицу 3.3).

Таблица 3.3 - Арифметические операторы языка Verilog.

Оператор	Название	Пример
+	Сложение	2 + 2 = 4
-	Вычитание	3 – 1 = 2
*	Умножение	2 * 3 = 6
/	Деление	5 / 2 = 2
%	Остаток	5 % 2 = 1
**	Возведение в степень	2**3 = 8
-	Унарный минус (смена знака)	-(1) = -1

3.8.3 Логические операторы (logical)

Логические операторы служат для получения логических значений (ИСТИНА/ЛОЖЬ). Результат выполнения этих операций всегда является однобитным, т.е. 1 или 0.

К логическим операторам относятся:

&& - логическое И;

|| - логическое ИЛИ;

! – логическое отрицание (НЕ).

Отличие побитных и логических операций можно наглядно проиллюстрировать следующим примером:

```
1111 & 1111 = 1111
```

– результат содержит столько же разрядов, сколько операнды;

```
1111 && 1111 = 1
```

- результат представляет собой однобитное логическое значение

.

Необходимо обратить внимание, что объект, принимающий результат, должен быть *одноразрядным*, чтобы избежать расширения результата нулями до разрядности приемника. Игнорирование этого требования может привести к логическим ошибкам, которые трудно обнаружить, если разработчик ошибочно посчитает, что все разряды приемника принимают одинаковое значение. Например, если сначала выполнить:

```
a = 8'b0000_0001;
b = 8'b0000_0000;
reg [7:0] flag;
assign flag = a || b;
```

то при попытке выполнить проверку вида

```
11110000 & flag;
```

результат будет равен 00000000, несмотря на то, что формально оба операнда должны трактоваться как логическая ИСТИНА, поскольку в обоих хотя бы один разряд равен 1.

Вообще, следует осторожно относиться к представлению логических значений многоразрядными числами, поскольку при несовпадении положений единичных разрядов логическое И над такими числами может дать в результате 0, то есть ЛОЖЬ. Например:

```
0011 & 1100 = 0000,
```

хотя оба операнда содержат хотя бы одну единицу и трактуются как ИСТИНА.

3.8.4 Операторы отношения (relational)

Операторы отношения служат для выполнения операций сравнения между операндами.

> - «больше»;

< - «меньше»;

>= - «больше или равно»;

<= - «меньше или равно».

Результатом сравнения является логическое (однобитное) выражение ИСТИНА или ЛОЖЬ.

3.8.5 Операторы равенства/тождества (equality)

К операторам проверки равенства и тождества относятся:

== - «равно»;

!== - «не равно»;

=== - «тождественно равно»;

!== - «тождественно не равно».

Несколько необычные варианты операций «тождественно равно» и «тождественно не равно» проистекают из возможности появления значений z и x (unknown) при моделировании.

Если хотя бы один из битов сравниваемых операндов равен этим значениям, обычное сравнение выполнить не удается, и результатом операции == будет x (неизвестно).

Однако можно произвести «буквальное сравнение» с помощью операции ===, которая дает истинный результат, если в каждом бите обоих операндов находятся строго одинаковые символы, включая z и x. Это можно проиллюстрировать примером:

```
if (a === 000x) …
```

операторы, показывающие, что младший разряд равен неизвестному значению

Можно добавить, что операторы проверки тождественности имеет смысл использовать только при моделировании, где они полезны для подробной диагностики состояния моделируемой системы и появляющихся в ней ошибок. Приведенный пример позволяет представить возможности глубокой диагностики появляющихся логических ошибок, когда в модели будет проверяться не только появление значений «неизвестно», но еще и выявляться разряд, в котором они появляются.

3.8.6 Операторы свертки (reduction)

Операторы свертки являются компактной формой записи операторов, которые выполняются попарно над разрядами одного операнда. Иными словами, операция применяется к двум младшим разрядам многоразрядного операнда, над ее результатом и следующим разрядом опять проводится такая же операция, и так далее вплоть до самого старшего разряда. Verilog допускает следующие операции свертки

& И;

~& - И-НЕ;

| ИЛИ;

~| ИЛИ-НЕ;

^ ИСКЛЮЧАЮЩЕЕ ИЛИ;

~^ ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ.

Практическое применение операторов свертки заключается, например, в проверке многоразрядного числа на ноль:

```
wire [7:0] a;
reg non_zero;
assign non_zero = |a;
```

При выполнении свертки операция ИЛИ будет сначала применена к разрядам 0 и 1 сигнала `a`. Полученный однобитный результат будет «свернут» с разрядом 2, и так далее. В соответствии со смыслом операции ИЛИ, наличие единицы в любом из разрядов a даст единицу в качестве результата свертки.

Аналогично, `&a` даст в результате 1 только в том случае, если *все* разряды `a` равны 1. Операция `^a` при этом вычислит *четность* `a`, то есть ответит на вопрос, является ли количество разрядов, установленных в единицу, четным.

3.8.7 Условный оператор (conditional)

Единственный триарный условный оператор Verilog является более короткой формой традиционного условного оператора `if/then`. Пример такого оператора:

```
assign q = sel ? a : b;
```

В этом примере выходу **`q`** будет присвоено значение **`a`**, если сигнал **`sel`** представляет собой логическую ИСТИНУ, и сигнал **`b`** в противном случае.

Удобно использовать такую форму условного оператора для компактного описания буфера с тремя состояниями:

```
assign q = en ? data : 1'bz ;
```

3.8.8 Операторы конкатенации/повторения (concatenation/replication)

Операторы конкатенации служат для группировки сигналов в шины. Операторы повторения (*replication*) являются удобной формой объединения повторяющихся сигналов.

Например, если **`a, b, c, d`** представляют собой одноразрядные сигналы, то выражение `{a, b, c, d}` создаст четырехбитный сигнал, составленный из «сцепленных» разрядов `a, b, c, d` соответственно. Здесь символы "`{ }`" служат для оформления оператора конкатенации.

Оператор повторения (репликации) является разновидностью оператора конкатенации. Например, выражение:

```
{4{a}}
```

полностью аналогично записи

```
{a, a, a, a},
```

т.е. число 4 обозначает количество повторений сигнала, указанного во внутренних фигурных скобках.

3.8.9 Операторы сдвига (shift)

Операторы сдвига являются аналогами подобных операторов, используемых в языках программирования. Они позволяют записывать манипуляции с шинами, заключающиеся в сдвиге их значений на один или несколько разрядов.

Существуют следующие операторы сдвига:

>> - логический сдвиг вправо;
<< - логический сдвиг влево;
>>> - арифметический сдвиг вправо;
<<< - арифметический сдвиг влево.

Все операторы имеют два операнда – сдвигаемая шина и число разрядов, на которые производится сдвиг. Например, сдвиг шины a на 2 разряда вправо:

```
assign q = a >> 2;
```

Отличием логического и арифметического сдвигов является поведение старшего разряда. Дело в том, что операции сдвига являются удобным способом умножения и деления на целые степени двойки – 2, 4, 8, 16 и т.д. Действительно, рассматривая двоичное число 0001, нетрудно убедиться, что его сдвиг влево на 1 разряд даст число 0010 (т.е. 2_{10}), далее 0100 (4_{10}), 1000 (8_{10}). Рассмотрим теперь представление чисел в дополнительной двоичной форме:

0_{10} -1_{10} = -1_{10} → 00000000_2 – 00000001_2 = 11111111_2 = 255_{10}

-1 -1 = -2 → 11111111_2 – 00000001_2 = 11111110_2 = 254_{10}

Можно предположить, что операция деления -2/2 с помощью сдвига на один разряд вправо должна давать правильный результат. Если воспользоваться логическим сдвигом, при котором в освобождающиеся разряды помещается 0, то 11111110 >> 1 даст в результате 01111111_2, что соответствует десятичному числу 127. Для беззнаковой арифметики этот результат является правильным, т.к. 254/2 = 127. Однако при трактовке числа 11111110_2 как числа -2 в дополнительной двоичной арифметике результат некорректен.

Для решения этой проблемы используется операция арифметического сдвига вправо, который отличается тем, что старший разряд сдвигаемого числа сохраняется. В этом случае оказывается, что 11111110 >>> 1 = 11111111, т.е. -2/2 = -1.

3.9 Приоритет операторов

При наличии нескольких операторов в выражении они выполняются с учетом приоритета. Наивысшим приоритетом обладают унарные операторы, низшим – опера-

торы конкатенации сигналов.

Список операторов Verilog в порядке убывания приоритета приведен ниже:

унарные и операторы редукции	+ - ! ~ & ~& | ~| ^ ~^ ^~
возведение в степень	**
умножение, деление, остаток	* / %
сложение, вычитание	+ -
сдвиг	<< >> <<< >>>
операторы отношения	< <= > >=
сравнение	== != === !===
логическое	И &
побитное	ИСКЛЮЧАЮЩЕЕ ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ ^ ~^
побитное	ИЛИ |
логическое	И &&
логическое	ИЛИ||
условный оператор	? :
операторы конкатенации и репликации	{} {{}}

При необходимости изменения порядка вычисления следует использовать скобки. Кроме того, расстановка скобок в ряде случаев помогает получить более эффективную схемотехническую реализацию при синтезе.

3.10 Процедурные блоки

Процедурные блоки предназначены в основном для моделирования цифровых устройств на Verilog. Однако блоки always могут быть также и синтезируемыми, и предназначены для описания синхронных (тактируемых) схем. Блок always имеет следующий синтаксис:

```
always @(<список чувствительности>)
begin
...
<операторы>
...
End
```

Список чувствительности (*sensitivity list*) представляет собой список сигналов, изменение которых должно приводить к выполнению операторов, перечисленных внутри «операторных скобок» `begin/end`. При этом символ “`@`” (читается как «*at*») вместе с ключевым словом `always` образует фразу вида «*всегда при () выполнять*».

Можно, например, реализовать с помощью блока `always` обычное комбинаторное выражение:

```
always @(a, b)
begin
  q = a & b;
end
```

Такая запись эквивалентна выражению непрерывного присваивания:

```
assign q = a & b;
```

С точки зрения моделирования, различие состоит в том, что для оператора `assign` повторный расчет выражения производится при изменениях любого из сигналов, участвующих в выражении, а для блока `always` – только для сигналов, перечисленных в списке чувствительности. Для приведенных выше примеров результат будет одинаковым, однако возможны ситуации, когда сокращение списка чувствительности до необходимого минимума способен ускорить процесс моделирования за счет отказа от повторного перерасчета выражений, которые в конечном итоге не влияют на работу схемы. С другой стороны, рассмотрим запись:

```
always @(a)
begin
  q = a & b;
end
```

Несмотря на то, что будет синтезирован вентиль 2И, при его моделировании значение выхода не будет рассчитываться при изменениях сигнала `b`, поскольку его нет в списке чувствительности. Следовательно, результаты моделирования окажутся несоответствующими реальному поведению такого устройства.

Основное использование блока `always` в синтезируемых схемах – реализация синхронных схем, простейшим примером которых является *D-триггер* (*D-flip-flop*). Это устройство, имеющее вход данных `d` и выход данных `q`, а также тактовый вход `clk`. По фронту тактового сигнала выход принимает значение, равное действующему в этот момент на входе.

D-триггер, таким образом, является простейшим элементом памяти.

Описание D-триггера на Verilog с помощью процедурного блока `always`:

```
module dff(input clk,
                   input d,
                   output reg q);

always @ (posedge clk)
          q <= d;
endmodule
```

В списке чувствительности данного примера появилось ключевое слово `posedge`. Оно означает «положительный перепад» (*edge* – «край», «порог»), то есть пе-

реход от состояния 0 к состоянию 1. Этот момент времени соответствует, таким образом, переднему фронту сигнала `clk`.

При моделировании условие ***`posedge clk`*** *выполняется также и при переходах от x к 1 или от z к 1.*

По аналогии, условие наступления спада тактового сигнала (переход от 1 к 0) задается выражением `negedge clk`.

Внутри процедурного блока находится единственный оператор `q <= d`; При его выполнении выход `q` будет принимать значение, равное `d`. Поскольку выполнение этого оператора будет привязано к моментам положительных перепадов тактового сигнала, итогом синтеза для такого выражения будет D-триггер. Обратите внимание, что для сигнала `q` использован тип `reg`.

Комбинируя различные условия внутри процедурного блока, можно получить варианты триггера с различными особенностями поведения. К дополнительным сигналам, управляющим поведением триггера, обычно относятся:

`ce` (Clock Enable, разрешение счета), также обозначаемый как `en` (Enable, разрешение) – сигнал, разрешающий изменение состояния триггера. Высокий уровень на этом входе разрешает выполнение оператора q <= d.

`reset` (сброс) – сигнал сброса, устанавливающий триггер в состояние нуля. Различают синхронный сброс, выполняющийся по фронту тактового сигнала (обычно его и обозначают как `reset`), и асихронный, выполняющийся немедленно при появлении высокого уровня, который обычно обозначают как `clr` (clear, очистка).

`set` (установка) – сигнал установка, устанавливающий триггер в состояние единицы. Как и для сброса, различают синхронную установку (`set`), выполняющуюся по фронту тактового сигнала, и асинхронную (`preset`), выполняющуюся немедленно.

Кроме того, все управляющие сигналы могут иметь как высокий, так и низкий логический уровень, при котором условие их появления считается истинным. Такой уровень для сигналов также называют *активным*. Иными словами, если при появлении единицы на входе `reset` происходит сброс триггера, то говорят, что активным уровнем для входа `reset` является высокий, т.е. сигнал логической единицы. Активный низкий уровень часто используется в схемах, выполненных по технологии РТЛ или ТТЛ, где возможно использование схемы «монтажное ИЛИ». При такой схеме входной сигнал с помощью резистора подключался к положительной цепи питания, что обеспечивало «слабый» уровень логической единицы. При этом любой активный выход, который устанавливал уровень логического нуля, перекрывал единицу, установленную резистором.

Поведение триггера при одновременном действии нескольких сигналов с противоположным смыслом определяется описанием на Verilog. Например, если сначала проверить сброс, а потом установку, то сброс будет обладать приоритетом, поскольку при успешной проверке условия его наступления будет выполнено присвоение выходу нуля, и дальнейшие проверки выполняться не будут. Некоторые примеры описания триггеров показаны ниже.

Триггер с синхронным сбросом:

```
always @(posedge clk)
   if (reset) begin
      q <= 0;
```

```
    end else begin
       q <= d;
    end
```

Триггер с асинхронным сбросом и разрешением счета:

```
   always @(posedge clk or posedge reset)
      if (reset) begin
         q <= 0;
      end else if (ce) begin
         q <= d;
      end
```

3.11 Блочное и внеблочное присваивание в процедурных блоках

В приведенных выше примерах были использованы как оператор =, так и оператор <=. Они соответствуют так называемому блочному (в другом переводе – блокирующему) (обозначается знаком «=») и внеблочному (неблокирующему) (знак «<=») присваиваниям. Блочное присваивание (=) выполняется немедленно в том месте, где оно встретилось в тексте, при этом переменная сразу меняет свое значение, и это значение может быть несколько раз переприсвоено (изменено) внутри блока. Внеблочное присвоение (<=) изменяет значение переменной только в момент выхода из блока, и чтобы не вознкало конфликтов в одной ветви алгоритма для каждой переменной может быть только одна операция присвоения.

Рассмотрим, как эти свойства отражаются на поведении конструкций на языке Verilog. Для схемы, изображенной на рисунке 3.4. создадим тестовый пример, включающий в себя блочное присваивание значений входным сигналам:

```
initial
  begin
      a = 0;
      b = 0;
      c = 0;
      d = 0;
      #5 a = 1;  // *
      #10 b = 1; // **
      #15 c = 1; // ***
      #20 d = 1; // ****
  end;
```

Полученные при моделировании временные диаграммы показаны на рисунке 3.5

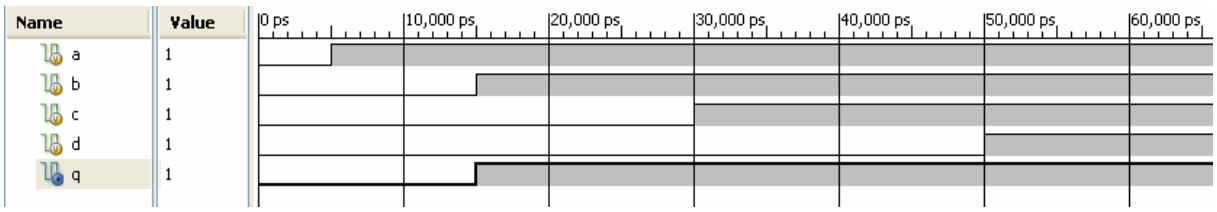


Рисунок 3.5 - Временные диаграммы сигналов в соответствии с моделью.

Блочный характер присваиваний проявился в том, что временные интервалы, указанные в строках, помеченных знаками «*», отсчитываются от предыдущей строки, в которой было использовано блочное присваивание.

Заменим теперь блочные присваивания на внеблочные:

```
initial
  begin
      a = 0;
      b = 0;
      c = 0;
      d = 0;
      a <= #5 1;
      b <= #10 1;
      c <= #15 1;
      d <= #20 1;
  end;
```

Полученные при моделировании временные диаграммы показаны на рисунке 3.6. Из него видно, что каждое внеблочное присваивание выполнено при выходе из блока и все они отсчитываются от момента начала моделирования.

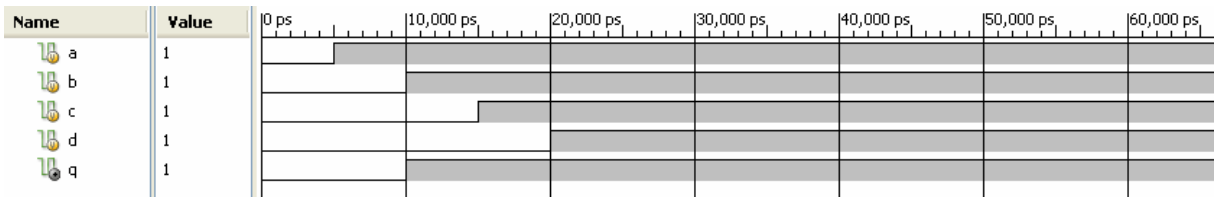


Рисунок 3.6 - Временные диаграммы сигналов в соответствии с моделью.

Рассмотрим схему, представляющую собой последовательное соединение двух триггеров.

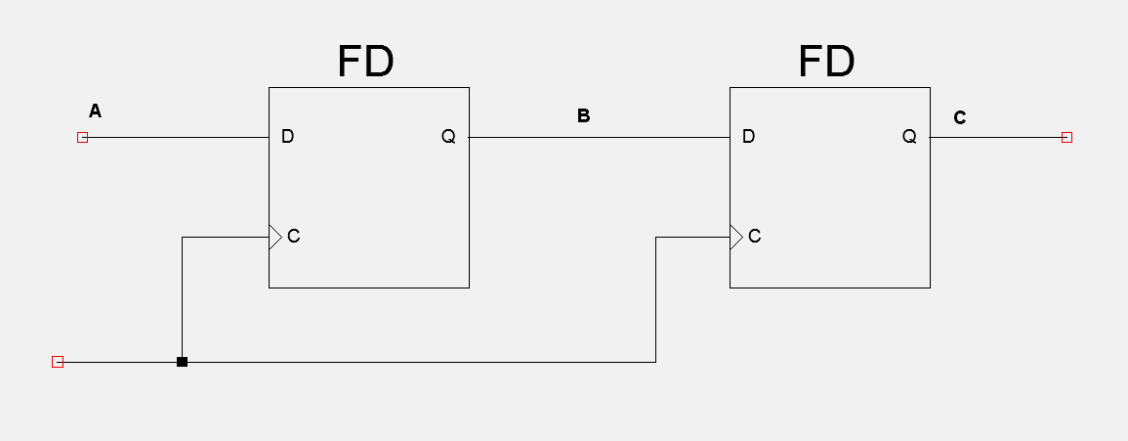


Рисунок 3.7 - Схема, состоящая из последовательно соединенных триггеров.

Опишем каждый триггер этой схемы в отдельном процедурном блоке.

```
always @ (posedge clk)
  b = a;
always @ (posedge clk)
  c = b;
```

С точки зрения синтеза, схема описана правильно, однако применение блокирующего присваивания приводит к тому, что корректность моделирования зависит от того, какой из процедурных блоков будет вычислен первым в процессе моделирования. Особенности реализации триггеров предполагают, что при одновременном приходе фронта тактового сигнала в первый триггер запишется сигнал `a`, а во второй – *старое* значение сигнала `b`. В то же время, если блочное присваивание `b = a` выполнится раньше, то модель покажет, что во второй триггер также запишется *новое* значение `b`, , т.е. сигнал, подаваемый по цепи `a`.

Одновременность процессов, происходящих в синхронных цифровых схемах, является очень важным принципом их работы, который требует отдельного внимания. Программистам известно, что для обмена значений, содержащихся в двух переменных, нельзя использовать конструкцию вида,

```
a = b;
b = a;
```

поскольку первый оператор заменит значение в переменной `a` на значение переменной `b`, и старое значение `a` станет недоступным. Для корректного обмена значений двух переменных требуется ввести временную переменную:

```
temp = a;
a = b;
b = temp;
```

Такой прием хорошо известен программистам, и может являться источником логических ошибок при проектировании цифровых схем. Рассмотрим схему, показанную на рисунке 3.8

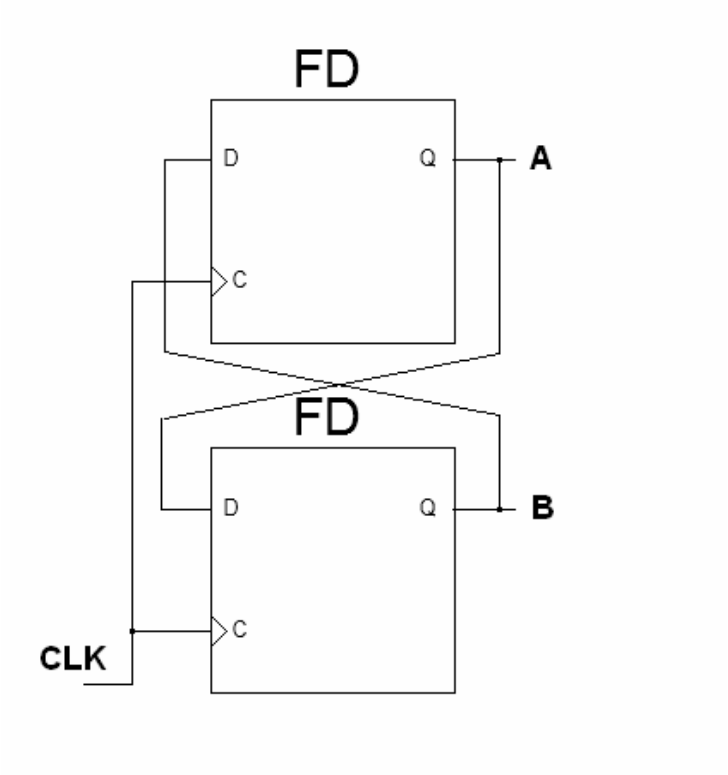


Рисунок 3.8 - Синхронная схема, состоящая из двух триггеров.

Принципиальный вопрос, на который требуется дать ответ, заключается в следующем: будут ли показанные на рис. триггеры обмениваться записанными в них значениями на каждом такте, или придут в какое-то фиксированное состояние, приняв одинаковое значение? Для ответа на него обратимся к временным диаграммам работы триггера (рис. 3.9).

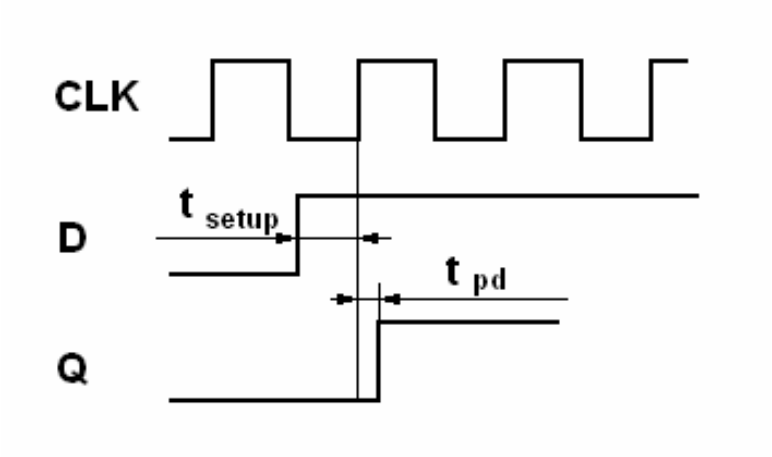


Рисунок 3.9 - Временные диаграммы работы D-триггера.

Как следует из порядка работы триггера, по фронту тактового сигнала его выход `q` принимает значение, равное значению на входе данных `d`. Однако, если обратиться к аппаратной реализации триггера, можно увидеть, что записывается то значение, которое присутствовало на входе `d` в течение некоторого времени t_{setup} *до* фронта тактового сигнала. Это время зависит от параметров конкретных полупроводниковых компонентов, и его минимально требуемое значение указывается в спецификации на цифровую микросхему. Оно называется *временем установки сигнала*. Существует также параметр, называемый *временем удержания сигнала* (t_{hold}), который задает время, в течение которого входной сигнал должен удерживаться на входе после фронта тактового сигнала. Для практических целей можно принять, что этот интервал времени равен нулю, то есть изменения уровня сигнала, присутствующие на входе триггера после прихода фронта тактового сигнала, уже не оказывают влияния на записанное значение.

Кроме того, от момента прихода фронта тактового сигнала до появления записанного значения на выходе триггера проходит некоторое время t_{pd}, называемое *задержкой распространения* (от propagation delay – «задержка распространения»). Сравнив временные диаграммы работы триггера со схемой, показанной на рисунке 3.9, нетрудно убедиться, что каждым тактом триггеры будут записывать старое значение своего соседа, то есть *обмениваться* состояниями. Очевидно, для этого требуется, чтобы оба триггера использовали один и тот же тактовый сигнал.

Подобный принцип работы цифровых устройств существенно отличается от принципов работы алгоритмических языков программирования, которые выполняют операции последовательно, и старое значение переменной оказывается недоступным после записи в нее нового значения (конечно, если язык программирования не обеспечивает специальные меры для сохранения предыдущего значения). Поэтому в языке Verilog корректнее пользоваться внеблочными присваиваниями, в которых автоматически запоминается предыдущее состояние переменной.

```
always @ (posedge clk)
  b <= a;
always @ (posedge clk)
  c <= b;
```

В приведенном примере порядок выполнения процедурных блоков не оказывает влияния на возможность считывания предыдущих значений, поскольку оба присваивания выполняются одновременно при выходе из блоков и для них используются старые (определенные ранее) значения переменных. При этом результаты моделирования окажутся корректными при любом порядке выполнения присваиваний.

Можно запомнить несложные практические правила:

- блочные присваивания используются для чисто комбинаторной логики;
- внеблочные присваивания используются для синхронной (тактируемой) и смешанной логики.

3.12 Управляющие структуры

Рассмотрим управляющие структуры, используемые в языке Verilog. Они используются как для моделирования, так и для синтеза, но часть из них требуют их упоминания только внутри процедурных блоков.

3.12.1 Условный оператор if/then

Условный оператор имеет в Verilog одну из следующих форм:

```
if  (<условие>)  <оператор>;
```

Если требуемое действие не может быть выражено одним оператором, используются «операторные скобки» `begin/end`.

```
if  (<условие>)
begin
<оператор1>;
<оператор2>;
     …
<оператор_n>;
end
```

Вторая форма условного оператора подразумевает использование ключевого слова `else` («иначе»):

```
if  (<условие>)  <оператор1>;
else < оператор2 >;
```

Такая форма используется при необходимости проверки множественных условий. При этом образуются вложенные операторы `if/else`:

```
if (<условие1>)  <оператор1>;
  else  if(<условие2>)  <оператор2>;
    else  if(<условие3>)  <оператор3>;
      else  if(<условие4>)  <оператор4>;
        else <оператор5>; // ни одно из условий не выполняется
```

Рассмотрим пример реализации мультиплексора **4-в-1**. Такой мультиплексор имеет мультиплексируемые входы `a, b, c, d`, вход селектора `sel` и выход данных `q`. Процедурный блок с использованием вложенных операторов `if/else` приведен ниже:

```
always @  *
if        ( sel == 2'b00 )  q = a;
 else if  ( sel == 2'b01 )  q = b;
 else if  ( sel == 2'b10 )  q = c;
 else if  ( sel == 2'b11 )  q = d;
 else                      q = 1'bx;
```

В приведенном листинге интерес представляет оператор по умолчанию, который присваивает выходу значение x («неизвестно»). По правилам цифровой схемотехники, использование этой строки излишне, поскольку на двухразрядном входе `sel` всегда будет присутствовать одна из комбинаций уровней, перечисленных в четырех вариантах условий. Однако с точки зрения моделирования в САПР, возможны также комбинации Таким образом, перечисленные варианты без последней строки не будут считаться исчерпывающим перечислением состояний такой схемы, и она не сможет быть реализована с помощью только комбинаторной логики (для чисто комбинаторной схемы требуется, чтобы ее выходное состояние было однозначно определено в каждый момент времени и для любой комбинации входных сигналов). Для устранения этого недостатка САПР введет в такую схему специальный элемент: *защелку* (latch) – схему удержания последнего состояния. В результате при отсутствии последней строки в схему буде введен элемент, который не предусматривается разработчиком, вносит дополнительную задержку и требует аппаратные ресурсы для своей реализации.

Для ПЛИС с архитектурой FPGA использование защелок не рекомендуется производителями, поскольку они строятся на базе триггера и мультиплексора. При выборе между триггером и защелкой следует выбирать триггер, а появление защелки в комбинаторной схеме, вызванное неполным описанием условий, является стилистической ошибкой.

Таким образом, лишняя на первый взгляд строка, явно задающая действия по умолчанию, позволяет избежать появления защелок, которые не требуются по смыслу работы схемы. Кроме того, при моделировании появление значения «неизвестно» позволяет быстро диагностировать проблемы при формировании тестовых воздействий, поскольку выполнение ветки `else` возможно только в том случае, если в модели не удалось обеспечить правильные логические уровни для сигнала `sel`.

Для сравнения можно привести следующий текст на Verilog:

```
 always @  *
if        ( sel == 2'b00 )  q = a;
 else if  ( sel == 2'b01 )  q = b;
 else if  ( sel == 2'b10 )  q = c;
 else                       q = d;
```

В данном примере в последней строке выходу присваивается значение входа d, поскольку предполагается, что если ни одно из предыдущих условий не выполняется, то значение селектора равно 11. С точки зрения синтеза это решение корректно, однако при моделировании оператор q = d будет выполняться и в случае, когда значение селектора не определено (т.е. равно «xx»). Таким образом, проблема в модели, заключающаяся в отсутствии корректного значения для сигнала-селектора, оказывается замаскированной, поскольку на выходе мультиплексора при моделировании все равно будет присутствовать какой-то сигнал, равный d. При этом будет неочевидно, появился этот уровень из-за действительного выбора входа d, или это явилось результатом неверно описанной модели. В то же время значение «неизвестно» окажет соответствующее влияние на входы, к которым подключен выход мультиплексора, сделав «неизвестным» результат работы соответствующих блоков, что даст возможность идентифицировать проблему и проследить ее до того блока, в котором и произошло некорректное задание тестовых воздействий. Можно также добавить, что состояния «неизвестно», «не инициализировано» обычно выделяются цветом при графическом построении временных диаграмм сигналов средствами моделирования, что позволяет быстро идентифицировать ошибки.

3.12.2 Оператор case

Оператор case соответствует оператору «выбор по переменной-селектору» и в языках программирования и в Verilog имеет вид:

```
always @ (<входы>)
begin
   case  (<выражение> )
      <вариант 1> : <оператор1> ;
      <вариант 2> : <оператор2> ;
      <вариант 3> : <оператор3> ;
      default :    <оператор по умолчанию> ;
   endcase
end
```

Здесь «выражение» - это выражение или переменная – селектор, на основании значения которого выполняется один из вариантов действий, перечисленных ниже в строках вида <вариант n> : <оператор n> ;

Аналогично оператору if, если после символа «двоеточие»

требуется указать более одного оператора, они должны быть указаны в «операторных скобках» begin/end.

Пример **мультиплексора 4-в-1** с использованием оператора case:

```
always @ *
begin
case  (sel)
   2'b00 : q = a;
   2'b01 : q = b;
   2'b10 : q = c;
   2'b11 : q = d;
   default : q = 1'bx;
endcase
end
```

Как и в примере с if, оператор default служит для предотвращения введения в схему защелки.

Допустимы множественные варианты выбора в одной строке, например:

```
2'b00, 2'b11 :
```

Сравнивая реализацию мультиплексора с помощью оператора if/else и case, можно сделать вывод об их логической эквивалентности. Действительно, описанные аналогичным образом, такие схемы будут давать одинаковые результаты при работе. Более того, для простых примеров схемная реализация, скорее всего, окажется идентичной (это зависит от особенностей конкретных САПР). Однако для микросхем программируемой логики рекомендуется при наличии возможности использовать именно оператор case, поскольку вложенные операторы if/else формируют «лестничную» комбинаторную структуру, при которой следующее условие может быть проверено только после того, как подтверждено *невыполнение* предыдущего. Такая структура обуславливает *дополнительную задержку* при распространении сигналов, проверяемых в самом первом условном выражении. В то же время для оператора case существуют шаблоны проектирования, известные САПР ПЛИС, которые позволяют, в частности, реализовать мультиплексоры с применением дополнительных ресурсов программируемых ячеек. Такая схема может оказаться быстрее и компактнее, чем схема, синтезированная из формально аналогичного оператора if/else.

Разумеется, оператор if/else не может быть заменен при проверке сложных условий, которые не могут быть сведены к проверке единственной переменной-селектора.

Таким образом, идеальный оператор case должен перечислять полный список возможных значений переменной селектора. Возможными негативными последствиями могут быть добавление защелки и реализация приоритетного дешифратора (если одно и то же условие задано в разных вариантах выбора, требуется проверка приоритета). Для устранения этих эффектов могут быть использованы директивы *full_case* и *parallel_case*, вводимые уровне RTL, которые устраняют защелки (*full_case*) и схемы проверки приоритета (*parallel_case*). Необходимо отметить, что нужный эффект от применения этих директив наблюдается только в том случае, если реализуемая схема в действительности обеспечивает полную и неперекрывающуюся дешифрацию всех возможных условий.

3.12.3 Оператор for

Оператор `for` в синтезируемых конструкциях используется в основном для создания множественных экземпляров схем. Необходимо сразу же подчеркнуть, что данный оператор *не предназначен* для создания счетчиков, или других схем, которые после их синтеза работали бы заданное количество тактов. Подобные задачи решаются в языках описания аппаратуры в совершенно ином стиле, и здесь не следует искать прямых соответствий с языками программирования. Рассмотрим пример:

```
module for_ex(
    input [3:0] a,
    input b,
    output reg [3:0] c
    );

integer i;


always @ (a, b)
begin

   for (i = 0; i < 4; i = i + 1)
       c[i] = b & a[i];
end

endmodule
```

Результат синтеза такой схемы показан на рисунке 3.10.

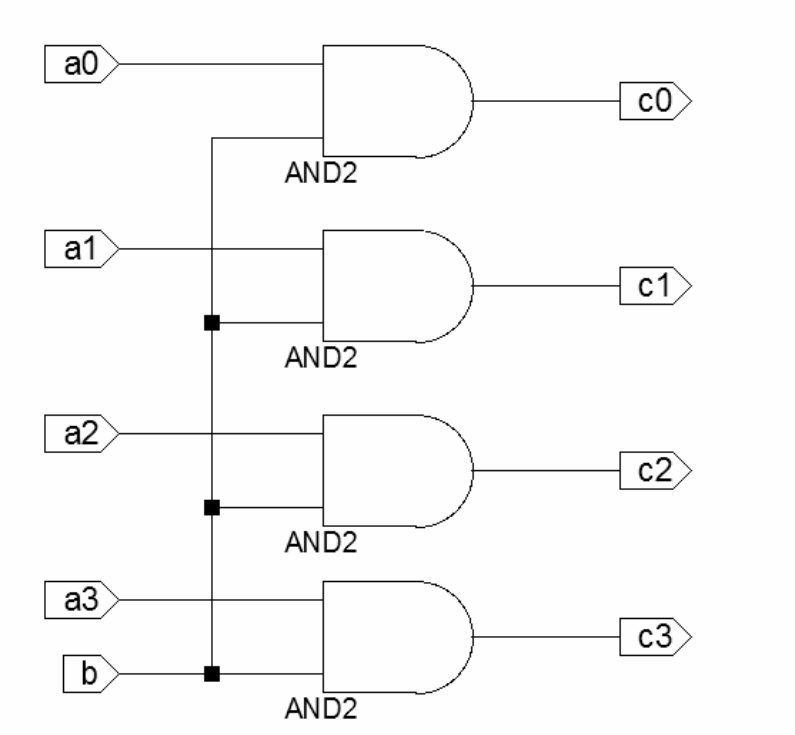


Рисунок 3.10 - Схема, синтезированная для примера с циклом for.

Из рисунка 3.10 видно, что оператор цикла оказал характерное воздействие прежде всего на САПР, заставив циклически синтезировать четыре одинаковых фрагмента цифровой схемы, различающихся индексами обрабатываемых сигналов. Такой вид использования, генерация множества экземпляров компонента, различающихся индексами сигналов, является наиболее естественной формой применения оператора `for`.

3.13 Задачи и функции

С целью упрощения разработки крупных проектов многие языки программирования предполагают различные варианты субпрограмм. Их применение позволяет повысить скорость и качество разработки за счет повторного использования кода и вынесения части функциональности в отдельные завершенные модули, что облегчает их отладку и модификации. Verilog предлагает две разновидности субпрограмм – задачи (`task`) и функции (`function`). Обе разновидности могут быть использованы как для моделирования, так и для синтеза.

Задачи отличаются от функций тем, что могут иметь произвольное количество входов, выходов и двунаправленных сигналов. Функция имеет входы, являющиеся ее аргументами, и возвращает единственный результат.

Задача имеет следующий формат:

```
task <имя задачи>
[формальные параметры]
[локальные переменные]
begin
  <операторы>
end;
endtask
```

Задача идентифицируется именем, которое задается после ключевого слова `task`. Впоследствии вызвать задачу можно, просто указав ее имя.

После объявления имени задачи можно указать формальные параметры, которые имеют тот же формат описания, что и порты модуля.

`in` – входные параметры, передаются в задачу;

`out` – выходы задачи, значения в них могут быть только записаны и возвращаются в вызывающий модуль;

`inout` – двунаправленные сигналы, передаются в задачу вызывающим модулем, могут быть прочитаны задачей и записаны, возвращаются в вызывающий модуль.

Если задача имеет формальные параметры, то при ее вызове следует передать ей фактические параметры в том же порядке, в каком формальные были объявлены в описании задачи:

```
task sum_task;      // определение задачи
input [7:0] a;
input [7:0] b;
output [7:0] sum;
begin
  sum = a + b;
```

```
end;
endtask
...
sum_task(op1, op2, result); // вызов задачи
```

При этом фактический параметр `op1` будет передан в задачу как вход `a`, `op2` – как вход `b`, а величина `sum` будет возвращена в вызывающий модуль в переменную `result`.

В стандарте Verilog-2001 возможно объявление формальных параметров в стиле языка Си:

```
task sum_task(input [7:0] a, input [7:0] b, output [7:0] sum)
```

Функции в Verilog, аналогично другим языкам, принимают произвольное число параметров, которые могут быть только входами. Функция возвращает единственный результат, имя которого совпадает с именем функции. Функция объявляется следующим образом:

```
function <имя функции>;
[входные параметры]
[локальные объявления]
begin
  <операторы>
end;
endfunction
```

Входные параметры функции объявляются так же, как и для задачи. Возможен Си-подобный вариант объявления:

```
function func_example1(input a, input b);
```

3.14 Организация проекта и параметризованные модули

При создании сложных проектов можно использовать ряд организационных и технических приемов, которые облегчают процесс разработки.

Директива `include предназначена для подключения текстов на Verilog, размещенных в сторонних файлах. Она оформляется следующим образом:

```
`include <«имя файла»>
```

Использование этой директивы равносильно вставке текста из упоминаемого файла непосредственно в месте упоминания директивы. Имя файла при этом задается в двойных кавычках, файл должен быть доступен из рабочего каталога в соответствии с общими правилами для данной ОС (т.е. файл, в котором упоминается директива, дол-

жен находиться в том же каталоге, или путь к подключаемому файлу должен быть указан явно).

Использование `include схоже с применением аналогичной директивы в языке Си, а подключаемые таким образом файлы Verilog можно сопоставить с заголовочными файлами Си (*header files*), в которых обычно размещаются константы, задачи и функции общего пользования. На имя подключаемого таким образом файла не накладывается специальных ограничений.

Предполагается, что в подключаемых файлах находятся объекты, облегчающие разработку. Такими объектами могут быть общие константы, используемые в различных модулях проекта (например, разрядность обрабатываемых данных или тактовая частота), функции общего назначения (например, алгоритмы кодирования и декодирования, функции и задачи, описывающие стандартные интерфейсы и узлы). При разработке подключаемых файлов рекомендуется тщательно планировать интерфейс размещаемых в них субпрограмм, поскольку частая смена интерфейсов заставит постоянно вносить соответствующие изменения в модули, которые используют эти субпрограммы. Также следует подробно документировать подключаемые файлы, сопровождать их примерами использования и тестами.

Параметры являются эффективным средством разработки масштабируемых модулей, т.е. модулей, для которых возможна быстрая смена разрядности, числа каналов, пределов счета и тому подобных численных характеристик, при сохранении алгоритмов работы. Параметр определяется с помощью директивы `define:

```
`define <имя_параметра> <значение>
```

Например:

```
`define DATA_WIDTH 8
```

Впоследствии при упоминании в тексте DATA_WIDTH вместо него будет подставлено значение 8.

Параметризованные модули являются достаточно привлекательными для разработки по многим причинам. Рассмотрим, например, реализацию регистров различной разрядности.

8-разрядный регистр:

```
module reg8( input clk,
             input [7:0] d,
             output reg [7:0] q);

always @ (posedge clk)
          q <= d;
endmodule
```

16-разрядный регистр:

```
module reg16( input clk,
              input [15:0] d,
              output reg [15:0] q);

always @ (posedge clk)
          q <= d;
endmodule
```

Из описания модулей можно видеть, что кроме имени, они различаются только разрядностью сигналов `d` и `q`. Нетрудно понять, что регистр любой разрядности будет описываться процедурным блоком, в котором будет находиться оператор `q <= d;`. Таким образом, для перехода к другой разрядности необходимо отредактировать только разрядность портов `d` и `q`.

Используем для управления разрядностью директиву `` `define ``:

```
`define DATA_WIDTH 8
module reg8( input clk,
             input [`DATA_WIDTH - 1:0] d,
             output reg [`DATA_WIDTH - 1:0] q);

always @ (posedge clk)
          q <= d;
endmodule
```

При объявлении портов теперь используется не прямое указание числа, а ссылка на определенный через `` `define `` параметр, названный `DATA_WIDTH` (буквально «ширина данных», или разрядность данных). Поскольку параметр в примере задан равным 8, выражение `[`DATA_WIDTH - 1:0]` эквивалентно `[7:0]`, что и требуется для 8-разрядного порта. Можно заметить, что редактирование единственной строки, где определен параметр, автоматически изменяет разрядности обоих сигналов – `d` и `q`, тогда как при ручном редактировании за приведением разрядности этих сигналов в соответствие друг другу необходимо следить отдельно. Представление требуемых величин в виде параметров особенно эффективно, если они используются в различных модулях – например, при разработке системы цифровой обработки сигналов, в которой каждый модуль производит очередное преобразование сигнала. При необходимости изменить разрядность обрабатываемых данных пришлось бы производить коррекции во всех модулях.

Verilog поддерживает также **локальные параметры**. Они имеют ограниченную область видимости (только в том блоке, в котором объявлены) и предназначены для того, чтобы не перекрывать имена ранее заданных параметров. Например, идентификатор `width` («ширина») является достаточно популярным при определении разрядности обрабатываемых данных. При его частом использовании в различных модулях может оказаться, что при их совместной трансляции в большом проекте значение параметра `width` окажется переопределенным одним из загруженных файлов. Для ограничения области видимости и используются локальные параметры, определяемые с помощью

ключевого слова localparam. Пример:

```
localparam x = 1;
```

Широкое использование параметров не является самоцелью, однако следует обратить внимание на те возможности по организации разработки, которые предоставляет параметризация. При создании модулей, которые используются во многих проектах, или для которых часто применяется частичная коррекция (например, изменение разрядности, пределов счета, начальных значений, порогов срабатывания и т.п.), настоятельно рекомендуется оформить часто изменяемые значения как параметры, вынеся их в отдельный текстовый блок (в пределах модуля или, при необходимости, во внешний загружаемый файл). В этом случае уменьшатся шансы случайно испортить исходные тексты модуля при внесении коррекции в величины, упоминающиеся по всему исходному тексту, поскольку исправления можно будет ограничить компактным текстовым фрагментом.

3.15 Условная генерация

Создание параметризованных модулей может также потребовать включения или выключения процесса синтеза для определенных фрагментов, или настройки числа создаваемых модулей. Это можно выполнить с помощью блока условной генерации, который оформляется с помощью ключевых слов generate и endgenerate.

Для создания множественных экземпляров какого-либо компонента в блоке generate необходимо также объявить специальную «генерирующую переменную» с помощью ключевого слова genvar. Переменная, созданная внутри блока generate, доступна только в этом блоке, тогда как переменная, объявленная вне его, доступна и другим подобным генерирующим циклам, размещенным в этом модуле.

Пример создания блока из 8 триггеров с помощью generate:

```
generate
genvar I;
for (I = 0; I < 8; I = I +1)
 begin: U    // метка обязательна
             // и задает именование создаваемых блоков
  DFF  DFF_INST (clk,
                 d[i],
                 q[i]);
end
endgenerate
```

Созданные триггеры имеют общий сигнал clk, а сигналы d и q подключаются к цепям d[0]..d[7] и q[0]..q[7] соответственно.

С помощью generate можно создавать компоненты, внутренние сигналы, задачи и функции, параметры, блоки непрерывного присваивания, процедурные блоки initial и always. Однако, таким образом невозможно создавать порты или изменять параметры существующих портов.

Конструкцию generate можно совмещать с условным оператором if для создания разных вариантов модуля:

```
integer platform_index;
…
generate
if (platform_index == 0) begin: U
   module_rtl …
end else
begin: V
  module_structural …
bnd
endgenerate
```

В показанном примере в модуле объявляется переменная `platform_index`, которая принимает значение, соответствующее номеру аппаратной платформы. Здесь предполагается, что для платформы, обозначенной индексом 0, не имеется готовых реализаций, описанных в виде низкоуровневого структурного представления, и для нее необходимо использовать высокоуровневое описание на Verilog. В зависимости от того, какое значение задано для `platform_index`, будет использовано то или иное описание для компонента.

Можно использовать также оператор **`case`**:

```
generate
  case(platform_index)
    0 : begin: U
    1 : begin: V
    2 : begin: W
    default:
  endcase
endgenerate
```

При разработке систем на базе ПЛИС с помощью условной генерации удобно описывать ресурсы, которые имеют разные детали реализации в разных семействах ПЛИС. Например, Spartan-3, Spartan-6, Virtex-4/5 и Virtex-6 имеют разные аппаратные примитивы для модулей формирования тактового сигнала. Такие модули настоятельно рекомендуются к использованию во всех проектах, но при смене семейства ПЛИС потребуется отредактировать проект, поскольку такие модули не имеют поведенческого описания, а добавляются с помощью *component instantiation* (в виде ссылки на библиотеку аппаратных примитивов). Можно также указать на вычислительные устройства, выполняющие операции «умножение с накоплением». В ранних вариантах Spartan-3 имеются только умножители формата 18x18, и аккумулятор требует реализации на базе логических ячеек. Семейства Spartan-3A DSP, Spartan-6 и Virtex-4 имеют также аппаратный 48-битный аккумулятор, а Virtex-5 и Virtex-6 – умножители формата 25x18 и 48-битный аккумулятор. Таким образом, если разработчик планирует использовать *instantiation*, он может создать модуль, использующий условную генерацию той или иной разновидности аппаратного примитива, в зависимости от используемого семейства ПЛИС.

Можно заметить, что параметр «семейство ПЛИС» оказывает глобальное влияние на весь проект, поэтому является подходящим примером для добавления в подключаемый файл, общий для всех модулей проекта. Детали реализации такого параметра (целочисленный индекс, текстовая строка) являются предметом отдельного обсуждения.

3.16 Моделирование на Verilog

Моделирование цифровых систем является важным шагом в маршруте их разработки. Возрастание сложности проектируемых устройств заставляет разработчиков тратить все больше времени на их моделирование. Целями моделирования могут быть как исследование алгоритмов работы проектируемого устройства, так и верификация характеристик, получаемых при его аппаратной реализации. В первом случае производится моделирование на верхних уровнях абстрагирования (т.е. преимущественно на поведенческом, и, возможно, RTL), а моделирование на физическом уровне призвано проверить возможность работы созданного устройства в заданных условиях эксплуатации (т.е. проверяется возможность работы на заданной тактовой частоте, с требуемыми длительностями сигналов, в заданном температурном диапазоне и т.д.).

При моделировании используется подход, основанный на «испытательном стенде» (*testbench*). Моделируемое устройство (в англоязычной литературе *UUT, Unit Under Test*) представляется своим синтезируемым кодом, а для проверки его поведения в различных условиях создаются описания тестовых воздействий («моделируемый код»).

Задание тестовых входных воздействий может быть выполнено на Verilog, с помощью модуля типа *Verilog Test Fixture* (см. рисунок 3.11).

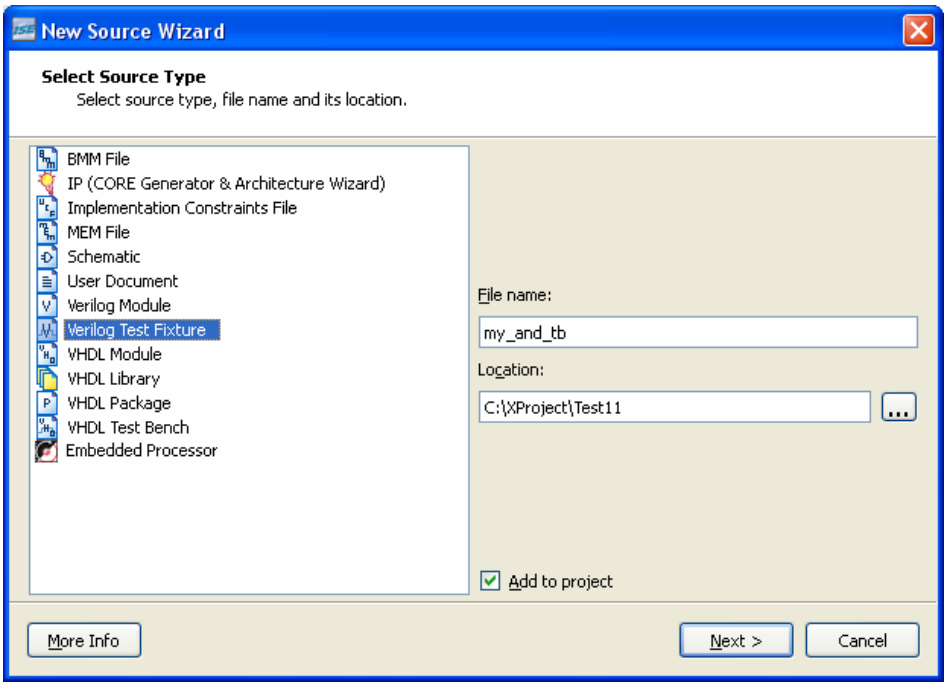


Рисунок 3.11 - Создание модуля, описывающего тестовые воздействия.

Для унификации обозначений обычно применяется сочетание символов ***tb*** (от *testbench*, «испытательный стенд»). Так, если модуль имеет название ***my_and,*** то с ним удобно сопоставить тестовый файл ***my_and_tb.***

При добавлении тестового файла в проект САПР ISE также запрашивает ассоциирование этого файла с одним из имеющихся модулей (рисунок 3.12).

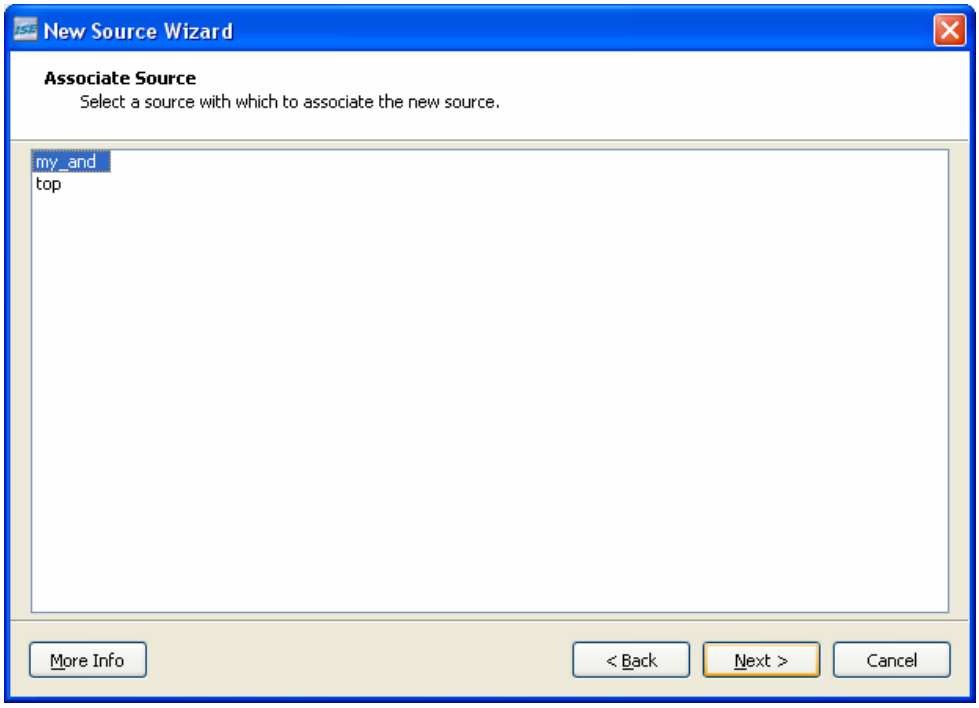


Рисунок 3.12 -Привязка модуля описания тестовых воздействий к синтезируемому модулю проекта.

Запрос на ассоциирование нужен для того, чтобы сгенерировать шаблон теста с автоматической привязкой к интерфейсу, как показано в листинге, приведенном ниже.

```
`timescale 1ns / 1ps
module my_and_tb;
	// Inputs
	reg a;
	reg b;
	// Outputs
	wire c;
	// Instantiate the Unit Under Test (UUT)
	my_and uut (
		.a(a),
		.b(b),
		.c(c)
	);
	initial begin
		// Initialize Inputs
		a = 0;
		b = 0;
		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here
	end

endmodule
```

Это описание необходимо модифицировать для получения требуемого эффекта от моделирования. Предположим, что требуется проверить поведение модуля при последовательной установке в логическую единицу сначала входа **a**, а затем входа **b**. Тогда раздел «инициализация входов» (`Initialize Inputs`) будет выглядеть так:

```
initial begin
        // Initialize Inputs
        a = 0; # 10; a = 1;
        b = 0; # 20; b = 1;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
End
```

На первом этапе моделирования проверяется поведенческая модель устройства. Переключение окна навигатора проекта в режим моделирования и запуск программы осуществляется так, как это показано на рисунке 3.13.

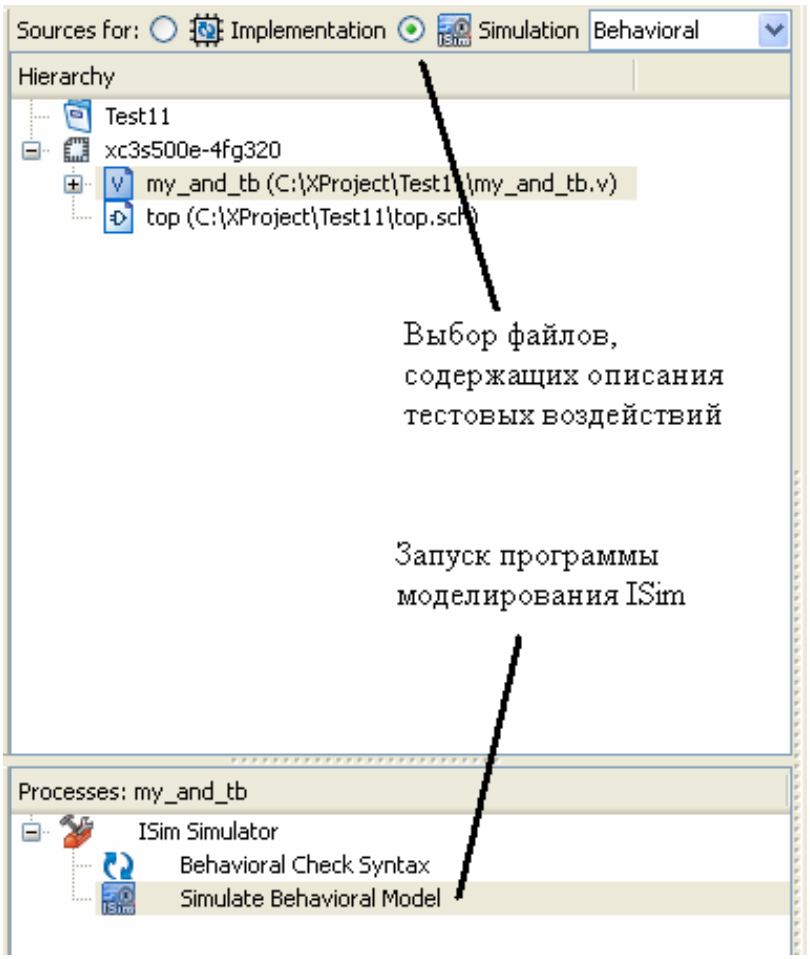


Рисунок 3.13 - Переключение между режимами работы САПР ISE.

На рисунке 3.14 видно, что в результате моделирования входные сигналы были установлены в соответствии с заданными тестовыми воздействиями. Задержки, заданные с помощью символа #, накапливаются, т.е. сигнал b получил значение 1 через время 20 нс не относительно начала моделирования, а относительно того момента времени, который был текущим после моделирования всех предыдущих процессов, включая и задержки. Поэтому последовательность действий, заданная тестом, интерпретируется следующим образом:

```
a = 0; установить a равным 0
# 10; ждать 10 единиц времени (единица времени равна 1 нс)
a = 1; установить a равным 0 (теперь текущее время равно
10 нс)
b = 0; установить b равным 0
# 20; ; ждать 20 нс (с учетом ранее накопленных 10 нс мо-
мент времени равен 30 нс)
b = 1; установить b равным 1
```

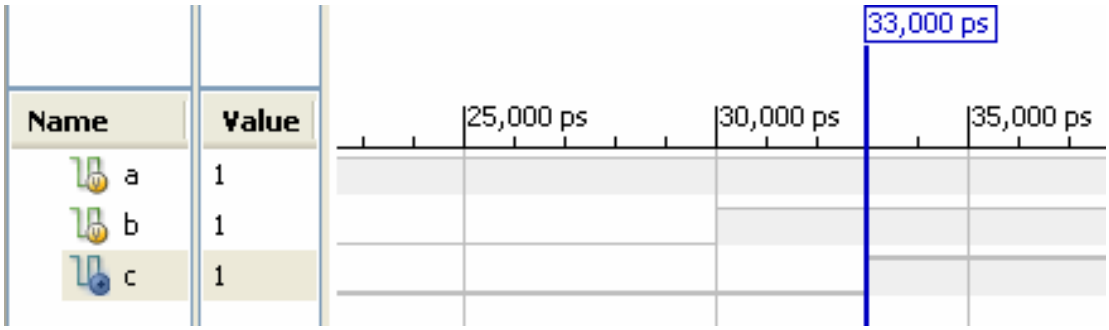


Рисунок 3.14 - Временные диаграммы при моделировании в поведенческом режиме

После этого, с учетом заданной задержки распространения сигнала в 3 нс, выход с примет значение 1 в момент времени 33 нс (30 нс до момента, в который оба входа приняли значение 1, плюс 3 нс на распространение сигнала до выхода).

Необходимо иметь в виду, что задержка в 3 нс, указанная в поведенческой модели, не является указанием САПР обеспечить заданную задержку. Напротив, приведение этого значения означает, что разработчик каким-то образом спрогнозировал задержку распространения, определил ее из технической документации, либо планирует рассмотреть, каким бы было поведение устройства, если бы задержки были равны указанным. Реальные задержки зависят от используемой аппаратной платформы, размещения компонентов на кристалле и особенностей выполнения трассировки.

В САПР ПЛИС ISE возможно проведение моделирования на физическом уровне, когда задержки распространения сигналов не принимаются равными тем, которые указаны в поведенческом описании, а рассчитываются, исходя из физических моделей компонентов ПЛИС и трассировки конкретного проекта. Для моделирования в таком режиме необходимо в выпадающем списке выбрать **Post-Route** перед запуском моде-

лирования. Результаты моделирования компонента my_and показаны на рисунке 3.15.

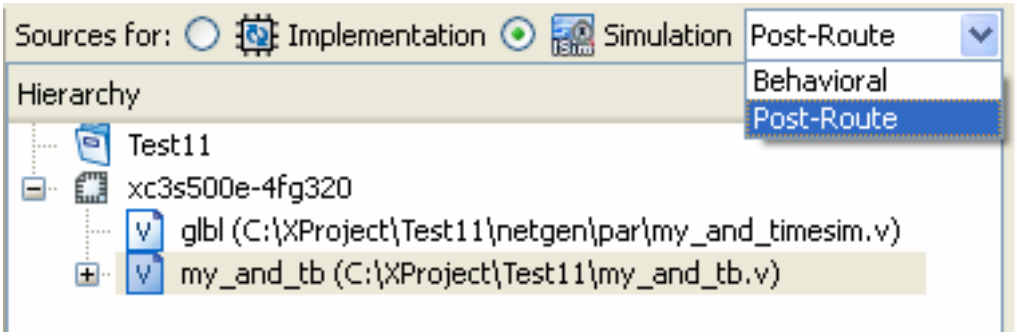


Рисунок 3.15 - Выбор режима моделирования

Результаты моделирования в режиме **Post-Route** приведены на рисунке 3.16. Можно убедиться, что задержка распространения сигнала оказалась равна 6,05 нс (для ПЛИС XC3S500E), а не 3 нс, как это предполагалось в поведенческой модели.

Величина 6,05 нс не является задержкой распространения через единственный логический вентиль, что было бы чрезмерно завышенным результатом для 90-нм элементной базы. Данная задержка определена с учетом влияния входных буферов для сигналов a и b и выходного буфера выхода c, подключенных к выводам ПЛИС.

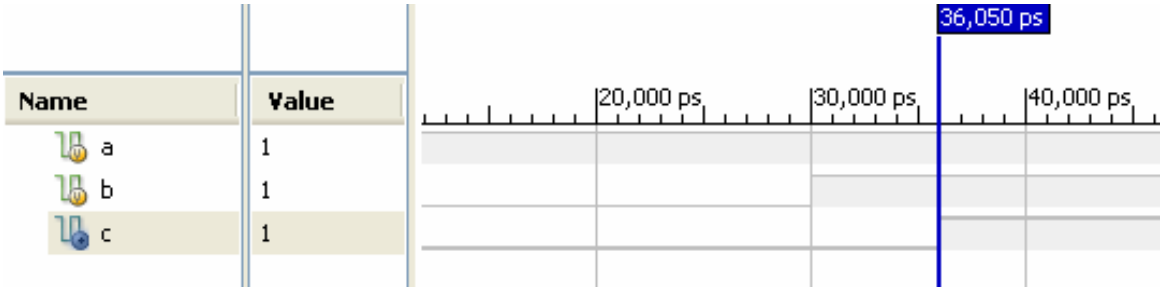


Рисунок 3.16 - Результаты моделирования модуля my_and в режиме Post-Route.

Для получения результатов **Post-Route** моделирования необходимо, как следует из названия, выполнить трассировку **(Routing)** проекта. Таким образом осуществляется привязка проекта к конкретной элементной базе. При этом выполняется анализ физических моделей компонентов, что требует существенно большего времени по сравнению с выполнением поведенческого моделирования. Однако такой результат существенно точнее, поскольку времена распространения сигналов рассчитываются по реальной трассировке, а не вводятся в модель из субъективных соображений.

При проектировании цифровых систем моделирование на физическом уровне обычно используется на завершающих этапах верификации проекта, когда требуется подтвердить не просто правильность выполнения преобразований, а соблюдение временных характеристик установления и распространения сигналов. В процессе отладки удобнее использовать моделирование на поведенческом уровне, поскольку оно произ-

водится существенно быстрее. В этом случае даже ориентировочные величины задержек позволяют иллюстрировать временной сдвиг сигналов друг относительно друга, что позволяет отлаживать архитектуру устройства.

Более сложные моделирующие конструкции можно создавать с помощью процедурных блоков `initial` и `always`. Процедурный блок `initial` выполняется однократно в процессе моделирования, в момент времени 0, а блок `always` – каждый раз, когда изменяется любой из сигналов, перечисленных в списке чувствительности.

Пример 1:

```
initial
begin
  clk = 1b'0;
  forever #10 clk = ~clk;
end
```

В этом примере с помощью ключевого слова `forever` организуется бесконечный цикл – через каждые 10 нс значение сигнала `clk` меняется на противоположное (подразумевается, что единицы времени для моделирования установлены равными 1 нс, как это обычно бывает).

Пример 2:

```
initial
begin
  q = 1b'0;
end

always @ (a,b)
begin
  q = a & b;
end
```

В примере выходному сигналу `q` присваивается значение, равное нулю. Далее следует процедурный блок `always`, который описывает логический вентиль И.

Основным преимуществом моделируемого подмножества Verilog является возможность создания моделей, описывающих задержки распространения сигналов. Это позволяет применять такие задержки к элементам, основываясь только на информации, указанной разработчиком модели, что существенно быстрее, чем получение этой информации путем анализа физической модели этого компонента. Таким образом, разработчик модели обязан обеспечить правильные величины задержек, но это компенсируется увеличением скорости моделирования.

Задержка указывается с помощью символа #. Например, для непрерывного присваивания:

```
assign #3 q = a & b;
```

#3 показывает, что задержка распространения сигнала составляет 3 нс (точнее, 3

«единицы времени», величина которых определяется директивой `timescale, и обычно равна 1 нс).

Для логических вентилей могут указываться три величины задержек, соответствующие следующим величинам:

- время перехода в высокий уровень (rise time);
- время перехода в низкий уровень (fall time);
- время отключения (turn off time).

Эти времена указываются после символа # в скобках, в порядке, приведенном в списке:

```
assign #(2,3,4) q = a & b;
```

Необходимо еще раз подчеркнуть, что приводимые таким образом задержки *используются только при моделировании*. Они *игнорируются* средствами синтеза, которые вместо этого могут рассчитать реальные задержки распространения, учитывающие используемые компоненты, соединяющие их проводники, условия работы и т.д.

Следующие конструкции не являются синтезируемыми и предназначены только для повышения удобства описания моделей.

Конструкция `wait`

Предназначена для синхронизации работы процедурных блоков. Например, в одном процедурном блоке может моделироваться установка сигнала, который используется в другом блоке:

```
wait (changed_signal)
  a = b;
```

Указанный блок (в примере состоящий из оператора `a = b`) будет выполняться каждый раз, когда изменится значение сигнала `changed_signal`.

Конструкция `while`

Представляет собой цикл с проверкой условия. Она имеет следующий синтаксис:

```
while(<условие>)
<оператор>;
```

При необходимости исполнять в цикле несколько операторов, как обычно в подобных случаях, используются «операторные скобки» `begin/end`.

Команда `forever` представляет собой бесконечный цикл:

```
initial
begin
   clk = 1'b0 ;
```

```
   forever  #10 clk = ~clk ;
end
```

Команда `repeat` повторяет цикл заданное число раз:

```
initial
begin
   counter = 0 ;
   repeat  ( 256 )
   begin
      $display ( "Counter = %d", counter);
      #10 counter = counter + 1;
   end
end
```

3.17 Создание отчетов и сообщений

Для формирования отчетов, требуемых разработчику, используется ряд моделирующих конструкций.

Команда **$display** имеет вид в соответствии с примером:

```
$display ("Во время %d сигнал a равен %b", $time, a)
```

При этом в отчет выводится указанное в кавычках текстовое сообщение в соответствии с заданными форматными модификаторами. В примере это `%d` и `%b` которые задают, соответственно, вывод числа в десятичном и в двоичном виде. Приведенное далее выражение `$time` соответствует времени модели, а переменная `a` указана в качестве примера. Кроме `$time`, можно использовать команду `$realtime`.

В процессе исполнения в отчете (консоли вывода средств моделирования) будет сформирована строка вида «Во время 5 сигнал a равен 1».

Команда `$write` аналогична `$display`, однако не выполняет перевод строки после завершения.

Команда `$strobe` также аналогична `$display`, но выводит сообщения по завершению текущего цикла моделирования.

Наконец, `$monitor` является еще одним аналогом `$display`, но с этой командой связаны также `$monitoroff` и `$monitoron`, которые, соответственно, выключают и включают вывод отчетов с помощью `$monitor`. Таким образом, с помощью этой команды можно задавать более подробные сообщения о состоянии модели, которые могут включаться по мере необходимости, и выключаться, когда их постоянный вывод перегружает отчет о моделировании.

Возможен вывод сообщений в текстовый файл, задаваемый пользователем. Для этого предназначены команды `$fdisplay`, `$fmonitor`, `$fstrobe`, `$fwrite`. Форматный модификатор `%t` отличается от `%d` тем, что выводит время в единицах, основанных на директиве `` `timescale ``. Формат вывода может также быть задан командой вида `$timeformat (-9, 3, "ns", 8)`. Здесь -9 означает десятичный порядок единицы измерения времени (в данном 10^{-9} означает наносекунду), 3 определяет число знаков, выводимых после десятичной точки, далее следует строка, которую необходимо вывести после значения времени, наконец, 8 задает минимальное количество символов для вывода.

Команда `$stop` приостанавливает моделирование и возвращает управление оператору. В это время можно, например, изменить значения входных сигналов и возобновить процесс моделирования.

Команда `$finish` завершает моделирование, закрывая все открытые файлы.

Форматные модификаторы для приведенных выше команд показаны в таблице 3.4.

Таблица 3.4 - Форматные модификаторы.

Название	Действие
`%h, %H`	Шестнадцатеричный формат
`%b, %B`	Двоичный формат
`%d, %D`	Десятичный формат
`%s, %S`	Текстовая строка
`%c, %C`	Символ ASCII
`%t, %T`	Время
`%o, %O`	Восьмиричный формат
`%v, %V`	Сила сигнала (power/strong/weak и т.п.)
`%e, %E`	Вещественное число в инженерном формате
`%f, %F`	Вещественное число в обычном формате
`%m, %M`	Иерархическое имя

При создании моделей на Verilog можно использовать файловые операции – заполнение массивов из текстовых файлов, открытие, создание, чтение и запись.

Команды `$readmemh` и `$readmemb` заполняют массив памяти из текстового файла, данные в котором представлены в шестнадцатеричном (`$readmemh`) или двоичном (`$readmemb`) форматах. Например, если объявлен массив `reg [7:0] array1 [0:1023]`, и имеется файл `array1_data.txt`, в котором размещены строки вида:

```
0000
1234
FA45,
```

то можно заполнить массив значениями из файла с помощью вызова:

```
$readmemh("array1_data.txt", array1);
```

Каждая строка соответствует одной ячейке массива памяти.

При использовании команды `$readmemb` в файле должны быть размещены двоичные значения.

Команда `$fopen` возвращает целочисленный индекс открытого файла. Этот индекс можно будет впоследствии указать в качестве аргумента одной из команд

$fdisplay, $fmonitor, $fwrite, $fstrobe. Например:

```
integer hLog;

initial
begin
hLog = $fopen("file1.log", "w");
$fdisplay ( hLog, "Текстовое сообщение, выводимое в
файл");
$fclose(hLog);
```

Команда **$fclose**, соответственно, закрывает ранее открытый файл.

Стандарты Verilog-95 и Verilog-2001 имеют различающиеся форматы вызова $fopen, и связанные с ними ограничения. Для Verilog-95 вызов $fopen требует только одного параметра – имени файла. В Verilog-2001 вторым параметром является режим доступа к файлу, которые перечислены в таблице 3.5.

Таблица 3.5 - Модификаторы режима доступа к файлу.

Обозначение режима доступа к файлу	Описание
"r" / "rb"	Открывает текстовый или двоичный файл для чтения
"w" / "wb"	Создает текстовый или двоичный файл для записи
"a" / "ab"	Открывает текстовый или двоичный файл для добавления
"r+b" / "rd+"	Открывает существующий двоичный файл для чтения и записи
"w+"	Создает новый текстовый файл для чтения и записи
"w+b" / "wb+"	Создает новый файл для чтения и записи
"a+"	Открывает существующий текстовый файл для добавления
"a+b" / "ab+"	Открывает существующий двоичный файл для добавления

Другим различием является максимальное количество одновременно открытых файлов. Для Verilog-95 это число равно 32, в Verilog-2001 число составляет 2^{30}. Старшие биты идентификатора файла зарезервированы для представления предопределенных идентификаторов файлов: stdin, stdout и stderr.

Кроме того, Verilog-2001 предлагает дополнительные функции для работы с файлами по сравнению с $readmemh и $readmemb, которые доступны в Verilog-95. Это следующие функции:

$fgetc(file) – читает из файла один символ;

$fungetc(char, file) – «возвращает в файл» символ, прочитанный ранее, делая его доступным для следующего вызова $fgetc;

$fscanf(file, переменные) – читает из файла переменные с возможным

использованием форматных модификаторов;

`$fgets(line, file)` – читает строку из файла;

`$fread` – читает из файла объекты в соответствии с заданным форматом.

Для функции `$fscanf` используется порядок вызова, аналогичный такой же функции в языке Си. Например:

```
$fscanf(hFile, "%b %d" bin_variable, decimal_variable);
```

прочитает из файла с индексом hFile две переменные, bin_variable и decimal_variable, причем значение для первой из них будет преобразовано из двоичного формата, а для второй – из десятичного. Данные в файле могут разделяться пробелами или запятыми. Форматные модификаторы для функции $fscanf приведены в таблице 3.6.

Таблица 3.6. - Форматные модификаторы для функции **fscanf**.

Модификатор	Описание
`%b`	Двоичное значение (допускаются состояния z, x, ?)
`%d`	Десятичное значение
`%o`	Восьмиричное значение
`%h, %x`	Шестнадцатиричное значение
`%e, %f, %g`	Значение в формате с плавающей точкой
`%t`	Формат представления времени
`%c`	Символ
`%s`	Строка
`%%`	Символ «%», преобразований не производится
`%u`	Битовое значение 0 или 1
`%z`	4-значное логическое значение: 0, 1, z, x

При чтении из файлов можно пользоваться символом EOF для проверки достижения конца файла:

```
while ( CHAR != `EOF ) begin
```

Литература к разделу 3:

3.1. Поляков А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры/ - М.: СОЛОН_Пресс, 2003. – 320 с.

3.2. Тарасов И. Е. Язык описания аппаратуры Verilog: Учебное пособие. – М.: МГТУ МИРЭА, 2011. – 132 с.

3.3. Стемпковский А.Л., Семенов М.Ю. Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: Учебное пособие. – М.: МИЭТ, 2005. -140 с.

4 Приемы проектирования на Verilog

В данном разделе будут рассмотрены приемы проектирования на Verilog применительно к ПЛИС с архитектурой FPGA. В качестве таковых рассматриваются микросхемы Xilinx семейств Spartan3, Spartan-6, Virtex-4, 5, 6.

4.1 Комбинаторная логика

Блоки комбинаторной логики добавляются с помощью оператора непрерывного присваивания `assign`:

```
assign q = a & b;
assign q = (a | b) ^ (c & d);
```

При необходимости приоритет операций регулируется скобками. Использование скобок также может быть полезно для управления процессом синтеза, поскольку в ряде случаев программы синтеза могут давать неоптимальные результаты. Например, результаты синтеза для выражения **q = a + b + c + d** и **q = (a + b) + (c + d)** могут оказаться различны. В первом случае будет синтезирована «лестничная» структура, где сигналы `a` и `b` пройдут последовательно через три сумматора, а во втором результаты окажутся чуть лучше, поскольку расстановка скобок заставит синтезатор создать два отдельных сумматора для выражений `a + b` и `c + d`, а затем сложить получившиеся выражения (рисунок 4.1). Приведенные рассуждения справедливы только для достаточно сложных выражений, поскольку сложение (и любые другие операции) над 4 входными линиями в любом случае могут быть реализованы в одной LUT.

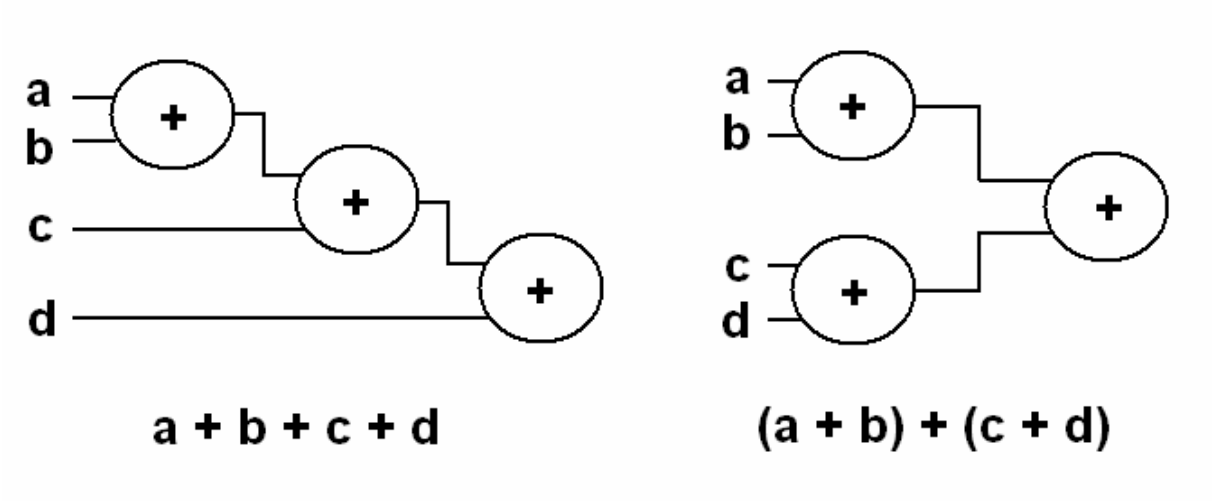


Рисунок 4.1 - Возможные результаты синтеза для эквивалентных арифметических выражений, различающихся расстановкой скобок.

4.2 Мультиплексоры

Мультиплексоры представляют собой компоненты, пропускающие на выход один из нескольких входов в соответствии с сигналом выбора (selector). Простейший вариант мультиплексора можно реализовать с помощью условного оператора:

```
assign mux = sel ? in1 : in0 ;
```

В приведенном примере описан **мультиплексор «2-в-1»**, который имеет, как следует из названия, два входа. Для того, чтобы выбрать требуемый вход, достаточно однобитного сигнала-селектора, поэтому порт `sel` является однобитным. Вообще, n-разрядный селектор достаточен для мультиплексора с 2^n входами.

Мультиплексируемые входы не обязаны быть одноразрядными. В показанном примере нет никаких сведений о разрядности сигналов in0, in0 и mux, которая в действительности может быть любой.

Мультиплексоры с большим числом входов можно реализовать на базе процедурного блока:

```
always @ *
begin
case  (sel)
   2'b00 : q = a;
   2'b01 : q = b;
   2'b10 : q = c;
   2'b11 : q = d;
   default : q = 1'bx;
endcase
end
```

Вариант с использованием `case` предпочтительнее использования вложенных операторов `if/else`, поскольку, как и в случае с арифметическими операциями, САПР ПЛИС имеет возможность использовать шаблоны проектирования, дающие оптимальные решения для мультиплексоров. Дополнительными ресурсами логических ячеек в этом случае являются их мультиплексоры – F5 и F6 для ПЛИС на базе 4-входовых логических генераторов (Spartan-3, Virtex-4) и F7, F8 для ПЛИС на базе 6-входовых логических генераторов (Spartan-6, все семейства Virtex).

4.3 Арифметические операции

Арифметические операции являются разновидностью операций, выполняемых с помощью комбинаторной логики, поскольку результаты вычисления арифметических выражений могут быть представлены в виде таблиц истинности, а следовательно, реализованы с помощью базовых логических операций. Например, суммирование двух однобитных чисел даст в результате следующие варианты:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 (1 переносится в следующий разряд)
```

Таким образом, результат сложения двух однобитных чисел может быть получен с помощью вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ, а бит переноса, если он необходим – с помощью вентиля И. Однако при проектировании для ПЛИС не следует заменять арифметические выражения на эквивалентные логические аналоги, поскольку в этом

случае САПР не сможет воспользоваться оптимальными шаблонами проектирования, которые, в частности, для операций сложения и вычитания, позволяют задействовать выделенные ресурсы логических ячеек – линии ускоренного переноса. Кроме того, построение многоразрядных сумматоров/вычитателей производится с применением шаблонов размещения – так называемых *Relationally Placed Macro* (RPM). Такие шаблоны обеспечивают оптимальное относительное размещение разрядов сумматора/вычитателя, которое, вероятнее всего, не будет достигнуто при синтезе описания сумматора, основанного на логических соотношениях между отдельными разрядами. Поэтому арифметические действия следует описывать с использованием символов соответствующих арифметических операций:

```
assign sum = a + b;
assign sub = a – b;
assign mul = a * b;
assign mac = mac + a*b;
```

Для операции умножения будет автоматически использован выделенный аппаратный умножитель. Последний пример представляет собой операцию «умножение с накоплением» (MAC, Multiply And Accumulate), которая широко используется в устройствах цифровой обработки сигналов. САПР ПЛИС распознают выражения представленного вида, реализуя их на базе блоков «умножение с накоплением», которые присутствуют во многих семействах ПЛИС.

4.4 Триггеры и регистры

Триггер представляет собой синхронный элемент и описывается процедурным блоком always. В списке чувствительности достаточно привести тактовый сигнал, и, если есть, вход синхронного сброса или установки, поскольку изменение других сигналов не оказывает немедленного влияния на выход триггера.

Описание D-триггера на Verilog с помошью процедурного блока:

```
module dff( input clk,
                  input d,
                  output reg q);
always @ (posedge clk)
      q <= d;
endmodule
```

Комбинируя различные условия внутри процедурного блока, можно получить варианты триггера с различными особенностями поведения. К дополнительным сигналам, управляющим поведением триггера, обычно относятся:

`ce` (Clock Enable, разрешение счета), также обозначаемый как `en` (Enable, разрешение) – сигнал, разрешающий изменение состояния триггера. Высокий уровень на этом входе разрешает выполнение оператора `q <= d`.

`reset` (сброс) – сигнал сброса, устанавливающий триггер в состояние нуля. Различают синхронный сброс, выполняющийся по фронту тактового сигнала (обычно его и обозначают как `reset`), и асихронный, выполняющийся немедленно при появле-

нии высокого уровня, который обычно обозначают как `clr` (clear, очистка).

`set` (установка) – сигнал установка, устанавливающий триггер в состояние единицы. Как и для сброса, различают синхронную установку (`set`), выполняющуюся по фронту тактового сигнала, и асинхронную (`preset`), выполняющуюся немедленно.

Кроме того, все управляющие сигналы могут иметь как высокий, так и низкий логический уровень, при котором условие их появления считается истинным. Такой уровень для сигналов также называют *активным*. Иными словами, если при появлении единицы на входе reset происходит сброс триггера, то говорят, что активным уровнем для входа reset является высокий, т.е. сигнал логической единицы. Активный низкий уровень часто используется в схемах, выполненных по технологии РТЛ или ТТЛ, где возможно использование схемы «монтажное ИЛИ». При такой схеме входной сигнал с помощью резистора подключался к положительной цепи питания, что обеспечивало «слабый» уровень логической единицы. При этом любой активный выход, который устанавливал уровень логического нуля, перекрывал единицу, установленную резистором.

Поведение триггера при одновременном действии нескольких сигналов с противоположным смыслом определяется описанием на Verilog. Например, если сначала проверить сброс, а потом установку, то сброс будет обладать приоритетом, поскольку при успешной проверке условия его наступления будет выполнено присвоение выходу нуля, и дальнейшие проверки выполняться не будут. Некоторые примеры описания триггеров показаны ниже.

Триггер с синхронным сбросом:

```
always @(posedge clk)
  if (reset) begin
    q <= 0;
  end else begin
    q <= d;
  end
```

Триггер с асинхронным сбросом и разрешением счета:

```
always @(posedge clk or posedge reset)
  if (reset) begin
    q <= 0;
  end else if (ce) begin
    q <= d;
  end
```

Различные варианты построения триггера находятся в справочной системе САПР ISE. Общей рекомендацией для элементной базы, выполненной с соблюдением технологических норм 90 нм и менее (Spartan-3, Virtex-4 и более новые семейства FPGA Xilinx), является использование только синхронных сигналов сброса и установки. Кроме того, желательно ограничиться только одним входом управления (сброс или установка). Не следует использовать сигнал сброса, если он предназначен только для инициализации триггера в начальный момент работы, сразу после программирования ПЛИС, поскольку начальные значения триггеров логических ячеек не являются случайными, а принудительно устанавливаются в процессе загрузки конфигурации. Таким

образом, отказ от лишнего входа сброса облегчает трассировку проекта, не занимая дополнительные программируемые ресурсы. Это же положительно сказывается на максимальной тактовой частоте, поскольку проект, перенасыщенный управляющими сигналами, требует большого количества трассировочных линий, которые в противном случае могли бы быть заняты другими цепями, для которых важно добиться малых задержек распространения.

Категорически не рекомендуется каким-либо образом разрывать сигнал тактовой цепи, например, с помощью логического вентиля И, управляя таким образом работой триггера. Для разрешения или запрещения срабатывания по фронту тактового сигнала следует использовать вход разрешения счета `ce`, который реализован в триггерах ПЛИС аппаратно. Подача на тактовый вход сигнала, прошедшего через логический генератор, ведет за собой более пологий фронт нарастания тактового сигнала, а также его отставание по времени от основного тактового сигнала, который подается на остальные элементы проекта. Это ведет к весьма негативному эффекту «гонки фронтов», когда триггер уже нельзя считать работающим синхронно с остальными компонентами проекта, и он может захватывать значение на входе данных, относящееся как к состоянию на предыдущем такте, так и уже обновленное состояние (что является неправильным с точки зрения построения синхронных схем). Этот эффект особо опасен тем, что его появление является непредсказуемым, а поведение неустойчиво – некорректное поведение может проявляться при повторной трассировке того же проекта с другими настройками САПР, для отдельных микросхем в большой партии, или как перемежающаяся неисправность в одной и той же микросхеме при изменении температуры и/или напряжения питания. Идентификация подобных эффектов и методы устранения являются в очень большой степени эмпирическими, и, в то же время, использование только глобальных тактовых сигналов кардинально решает данную проблему. При проектировании, таким образом, следует обращать внимание на то, чтобы все события для триггеров и элементов на их базе происходили строго по фронту тактового сигнала, который формируется соответствующим аппаратным блоком (DLL, DCM, PLL, CMT, MMCM для разных семейств FPGA) и подается в логические ячейки по глобальным тактовым линиям, имеющимся во всех семействах FPGA. Использование примеров, приведенных выше, соответствует данному требованию.

Регистр представляет собой многоразрядный триггер, и описывается тем же способом. Фактически, основной процедурный блок остается для регистра таким же, как и для триггера, а изменяется объявление портов.

```
module reg8( input clk,
             input [7:0] d,
             output reg [7:0] q);
```

Для регистра сохраняются те же правила проектирования, что и для триггера.

4.5 Сдвиговые регистры

Сдвиговый регистр представляет собой разновидность регистра, для которого по фронту тактового сигнала происходит сдвиг содержимого на один или несколько разрядов в какую-либо сторону.

Сдвиговый регистр можно представить следующим текстом на Verilog:

```
module shift_reg(
    input clk,
    input d_in,
    input ce,
    output [15:0] q
    );
reg [15:0] data;

always @(posedge clk)
begin
  if (ce) data <= {data[15:1], d_in};
end

assign q = data;

endmodule
```

В примере производится сдвиг содержимого внутреннего регистра `data` на один разряд влево. В младший разряд при этом помещается значение, подающееся на вход `d_in`, а старший разряд теряется. Работа сдвигового регистра управляется входом `ce`, низкий уровень на котором запрещает сдвиг.

Выход сдвигового регистра `q` является копией внутреннего сигнала `data`. Такое решение выбрано потому, что в строке `data <= {data[15:1], d_in}` производится как запись, так и чтение переменной `data`. Таким образом, эта переменная не может быть портом типа output, так как для такого порта допустима только операция записи.

Регистр, сдвигающий свое содержимое на регулируемое число разрядов, называется регистром (устройством) *барабанного сдвига* (*barrel shifter*). Это более сложное для реализации устройство, чем обычный сдвиговый регистр, поскольку требует дополнительного мультиплексора, из-за чего занимаемый в ПЛИС объем существенно возрастает. Регистр барабанного сдвига применяется, например, для сложения и вычитания чисел с плавающей точкой, где он производит нормализацию мантисс – сдвиг одной из мантисс на число разрядов, равное разнице двоичных порядков складываемых чисел. Поскольку для операций с плавающей точкой желательна высокая производительность, а обычный сдвиговый регистр может потребовать от 1 до N тактов для нормализации (где N – число двоичных разрядов мантиссы, которое для чисел двойной точности равно 53), применение устройства барабанного сдвига для этой цели является оправданным. Пример описания устройства барабанного сдвига:

```
reg [3:0] q;

 always @*
   case (sel)
     2'b00  : q = d;
     2'b01  : q = d << 1;
     2'b10  : q = d << 2;
     default: q = d << 3;
   endcase
```

Приведенный пример может быть дополнен регистром, поскольку в описании показан только асинхронный мультиплексор, сдвигающий входной сигнал `d` на 0, 1, 2 или 3 разряда.

4.6 Счетчики

Счетчик выполняет последовательное увеличение или уменьшение своего выходного значения по каждому тактовому импульсу. Простейший вариант счетчика:

```
reg [7:0] cnt;

always @ (posedge clk)
  cnt <= cnt + 1;
```

Поскольку для хранения значения счетчика выбрано 8 разрядов, счетчик будет осуществлять циклическое приращение своего значения от 0 до 255, после чего операция 255+1 опять сделает значение `cnt` равным 0.

Для счетчиков используются следующие возможности:

- регулируемое направление счета (up/down);
- возможность сброса;
- возможность загрузки.

Счетчик с регулируемым направлением счета приведен в следующем примере:

```
reg [7:0] cnt;

always @(posedge <clock>)
  if (up_down)
    cnt <= cnt  + 1;
 else
   cnt <= cnt - 1;
```

Счетчик имеет дополнительный управляющий вход up_down. Высокий логический уровень на этом входе означает счет на увеличение значения счетчика, а низкий – на уменьшение.

Счетчик с загрузкой имеет дополнительные входы – разрешение загрузки и загружаемые данные. Если на входе разрешения загрузки присутствует активный уровень, то счетчик принимает значение, заданное внешней шиной. Таким образом, с помощью этого интерфейса можно принудительно задать требуемое значение счетчика.

Пример счетчика с синхронным сбросом и загрузкой:

```
reg [7:0] cnt;

always @(posedge clk)
  if (reset)
    cnt <= 0;
  else if (ce)
    if (load)
      cnt <= d_in;
    else
      cnt <= cnt + 1;
```

Как и для остальных цифровых модулей на базе ПЛИС, рекомендуется исполь-

зовать синхронный сброс вместо асинхронного.

Двоичное кодирование является не единственно возможным алгоритмом работы счетчика. Вместо последовательного перебора двоичных значений в процессе работы возможно использование и других кодировок. Например, код Грея (Gray code) имеет то свойство, что для перехода к следующему значению достаточно изменить значение единственного разряда. Это полезно при обработке сигналов, в которых существует вероятность сдвига по времени между отдельными разрядами. Например, при переходе от двоичного состояния 0111 (7_{10}) к 1000 (8_{10}) из-за неодновременной смены разрядов может появиться состояние 0000, 1111 (или любое другое, в зависимости от порядка смены разрядов). В то же время подобный эффект при использовании кода Грея приведет к максимальной ошибке, равной 1.

Пример реализации счетчика, основанного на коде Грея:

```
parameter gray_width = 8;

reg  [gray_width-1:0] binary_value;
reg  [gray_width-1:0] gray_value;

always @(posedge clk)
  if (reset) begin
    binary_value <= {{gray_width{1'b0}}, 1'b1};
    gray_value <= {gray_width{1'b0}};
  end
  else if (ce) begin
         binary_value <= binary_value + 1;
         gray_value <= (binary_value >> 1) ^ binary_value;
  end
```

Другой разновидностью кодирования является LFSR (*Linear Feedback Shift Register*) – сдвиговый регистр с линейной обратной связью. Пример 4-разрядного LFSR:

```
reg [3:0] lfsr;

always @(posedge clk)
  if (reset)
    lfsr <= 4'h0;
  else if (ce) begin
    lfsr[3:1] <= lfsr[2:0];
    lfsr[0] <= ~^lfsr[4:3];
  end
```

Особенностью кодирования по LFSR является более быстрая смена состояний по сравнению с двоичным счетчиком, поскольку при двоичном кодировании возможна ситуация, когда прибавление единицы сменит все разряды, включая самый старший. На распространение бита переноса по всем разрядам двоичного счетчика тратится дополнительное время, по сравнению со сдвиговыми регистрами, в которых каждый разряд получает свое значение от соседнего разряда. Вдвигаемое в LFSR значение определяется в строке `lfsr[0] <= ~^lfsr[4:3]`, и зависит от разрядности счетчика.

Недостатком счетчика LFSR является меньшее число уникальных состояний по сравнению с двоичным счетчиком той же разрядности.

4.7 Делители частоты

Делитель частоты представляет собой вариант счетчика, который выдает на выходе сигнал, частота которого в заданное число раз меньше, чем входная тактовая частота:

```
module div_clk(
    input clk,
    output clk_out
    );

reg [7:0] cnt;

always @ (posedge clk)
  if (cnt == 199) cnt <= 0;
  else cnt <= cnt + 1;

assign clk_out = (cnt == 0) ? 1 : 0;

endmodule
```

В показанном делителе частоты используется внутренний счетчик с пределом счета 200 (счетчик может принимать значения от 0 до 199). Если достигнутое счетчиком значение уже равно 199, то следующий фронт тактового сигнала загружает значение 0, иначе значение увеличивается на 1. Выходной сигнал формируется асинхронно, путем проверки значения внутреннего счетчика на 0. Поскольку такое состояние наблюдается в течение только 1 периода тактового сигнала из каждых 200, выходная частота оказывается в 200 раз меньше входной.

В представленном примере выход делителя будет формировать короткие импульсы, длительностью 1/200 от общего периода выходной частоты. При необходимости получения импульсов с коэффициентом заполнения 50% можно заменить выражение для `clk_out` на:

```
assign clk_out = (cnt < 100) ? 1 : 0;
```

Сигнал, полученный с выхода такого делителя, не должен использоваться для тактирования других устройств ПЛИС. Его правильным использованием будет подключение к входу `ce` других синхронных модулей, для которых он будет разрешать счет на один такт из N, тем самым и обеспечивая деление частоты на N.

4.8 Таймеры

Таймер отличается от обычного счетчика наличием дополнительных возможностей по запуску, перезагрузке и останову. В примере показан простейший таймер, который обеспечивает задержку появления выходного сигнала на 200 тактов относительно появления входного сигнала `reload`. После достижения максимального значения таймер не повторяет цикл счета с нуля, а останавливается до появления сигнала `reload`. В таком режиме данное устройство может использоваться как *сторожевой таймер* (*watch-*

dog timer). Эта разновидность таймера предназначена для формирования предупреждений о том, что какое-то событие, вызывающее появление сигнала перезагрузки таймера, не происходило уже длительное время. Сторожевой таймер часто используется в микроконтроллерных системах управления, где его перезагрузка происходит в процессе выполнения программы. Отсутствие перезагрузки в течение времени, соответствующего полному циклу счета, свидетельствует о том, что микроконтроллер перестал периодически выполнять команды, приводящие к сбросу счетчика в ноль. Это, вероятнее всего, является следствием аппаратного или программного сбоя системы, и выход сторожевого таймера может являться сигналом аппаратного сброса микроконтроллера, или индикатором аварийного состояния системы. Пример таймера, сбрасываемого по сигналу `reload`, и прекращающего счет при достижении заданного состояния:

```
module timer(
    input clk,
        input reload,
    output timer_out
    );

reg [7:0] cnt;

always @ (posedge clk)
  if (reload) cnt <= 0;
  else if (cnt < 199) cnt <= cnt + 1;

assign timer_out = (cnt == 199) ? 1 : 0;

endmodule
```

4.9 Широтно-импульсная модуляция

Широтно-импульсная модуляция (ШИМ, также PWM – Pulse-Width Modulation) является эффективным способом цифрового управления силовыми системами. При этом регулирование мощности осуществляется путем управления отношением времени, в течение которого сигнал включен, ко времени, в течение которого он выключен. Такой способ управления обладает как минимум двумя достоинствами – он удобен для реализации в цифровой системе, и является энергоэффективным, поскольку в ключевом режиме работы управляющий элемент потребляет минимальную мощность (в закрытом состоянии ток через него стремится к нулю, а в открытом стремится к нулю падение напряжения, поскольку сопротивление переключающих элементов стремятся уменьшить).

Модуль ШИМ разрабатывается следующим образом. Входной сигнал `d` задает число тактов, в течение которых следует удерживать высокий логический уровень на выходе модуля. Внутренний сигнал `cnt` циклически перебирает состояния от 0 до максимального значения, которое может быть подано в качестве входного. Допустим, что внутренний счетчик является 8-разрядным (т.е. полный цикл счета содержит 256 тактов). Тогда при подаче на вход числа 10 на выходе такого модуля будет логическая единица в течение 10 тактов, а в течение остальных 246 – логический ноль. Увеличивая значение числа, поданного на вход `d`, можно увеличивать отношение времени включения выходного сигнала к общему времени цикла счета:

```
module pwm(
    input clk,
    input [7:0] d,
    output pwm
    );

reg [7:0] cnt = 0;

always @ (posedge clk)
  cnt <= cnt + 1;

assign pwm = (d > cnt)  ? 1 : 0;

endmodule
```

Для демонстрации работы созданного модуля используется следующая тестовая последовательность: формируется тактовый сигнал с периодом 20 нс, и на вход данных подается фиксированное значение 100_{10}. Тестовый модуль ***pwm_tb***:

```
module pwm_tb;
     // Inputs
     reg clk;
     reg [7:0] d;
     // Outputs
     wire pwm;
     // Instantiate the Unit Under Test (UUT)
     pwm uut (
          .clk(clk),
          .d(d),
          .pwm(pwm)
     );
     initial begin
          // Initialize Inputs
          clk = 0;
          d = 100;
          forever clk = #10 ~clk;
     end

endmodule
```

Временные диаграммы работы модуля в соответствии с поданными тестовыми последовательностями показаны на рисунке 4.2.

Рисунок 4.2 - Временные диаграммы работы модуля ШИМ.

4.10 Модули памяти

Память является важным и часто используемым элементом современных цифровых устройств. Она представляет собой массив однотипных ячеек, хранящих число фиксированной разрядности. По характеру выполняемых с ней операций память подразделяется на:

- постоянные запоминающие устройства (ПЗУ, также ROM – Read-Only Memory), которые хранят фиксированные данные без возможности их изменения;
- оперативные запоминающие устройства (ОЗУ, также RAM – Random Access Memory), допускающие изменение записанных данных.

Типы памяти также разделяют на энергозависимую (сохраняющую данные только при поданном питании) и энергонезависимую. Для ранних вариантов исполнения энергонезависимая память являлась практически синонимом ПЗУ, поскольку техническая реализация таких микросхем подразумевала хранение данных в ячейках с пережигаемыми перемычками, ультрафиолетовым стиранием и т.п. Все эти принципы исполнения обеспечивали сохранность данных при отключении питания, но и не позволяли изменять содержимое памяти без специального оборудования – например, память с ультрафиолетовым стиранием требовала, как следует из ее названия, источника УФ излучения для стирания данных и специального программатора. В настоящее время существуют устройства энергонезависимой памяти, которые допускают изменение содержимого без специального программатора. Например, flash-память, память с электрическим стиранием, память FRAM, другие перспективные типы памяти. Часть из них требует отдельного цикла стирания, а часть позволяет произвольно перезаписывать данные, как для микросхем ОЗУ.

По способу организации интерфейса модули памяти подразделяются на модули с асинхронным или синхронным интерфейсом, а также с параллельным или последовательным доступом. Наиболее простой вариант – память с параллельным асинхронным доступом. Графическое изображение такого модуля показано на рисунке 4.3.

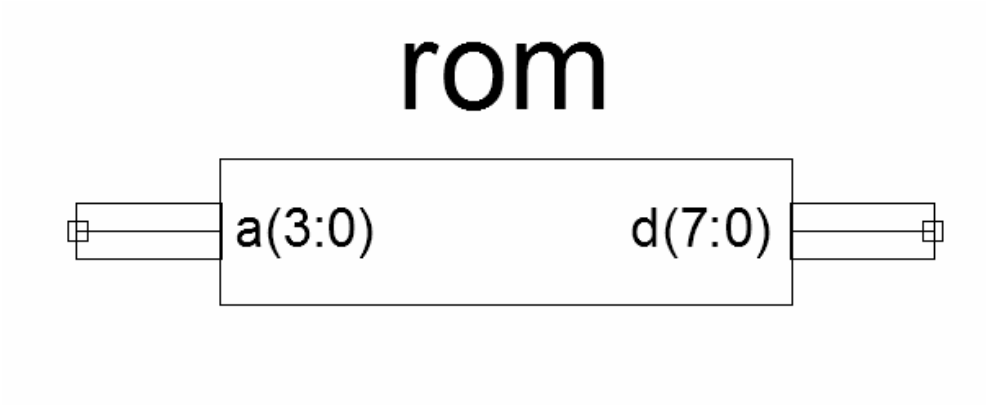


Рисунок 4.3 - Графическое изображение модуля памяти с асинхронным интерфейсом.

Постоянное запоминающее устройство с асинхронным интерфейсом может быть описано с помощью оператора **case**:

```
module rom(
    input [3:0] a,
    output [7:0] d
    );

reg [7:0] data;

always @(a)
case (a)
  0 : data <= 1;
  1 : data <= 3;
  2 : data <= 4;
  default : data <= 0;
endcase

assign d = data;
endmodule
```

При описании асинхронного ПЗУ с помощью оператора `case` используются строки вида `<addr> : <data>` - для каждого варианта адреса записывается то значение, которое хранится по этому адресу. Можно заметить, что при таком подходе сложно описать массивы памяти большого объема без применения средств автоматизации.

В ПЛИС модули ПЗУ небольшого объема обычно реализуются на базе логических генераторов программируемых ячеек. Ячейка с 6 входами (Virtex-5/6, Spartan-6) может хранить 64 бита, а с 4 – 16 бит. Необходимо иметь в виду, что из-за реализации в виде мелких блоков память на базе программируемых ячеек не может обеспечить высокие характеристики производительности.

Синхронный интерфейс памяти обеспечивает более высокую производительность, поэтому память такого вида используется чаще. Блочная память, размещаемая в FPGA, является памятью с синхронным интерфейсом. Отличием такого типа интерфейса является выполнение всех действий по фронту тактового сигнала.

Для памяти с возможностью чтения и записи (ОЗУ) используются следующие дополнительные сигналы:

`din` – данные для записи;

`we` – разрешение записи (Write Enable).

Память такого типа работает следующим образом: если по фронту тактового сигнала активен сигнал `we`, то производится запись данных `din` в ячейку памяти с адресом `addr`. Иначе производится чтение из памяти, и на выходе `dout` появляется содержимое ячейки памяти с адресом `addr`. Пример описания на Verilog памяти с произвольным доступом с синхронным интерфейсом:

```
module ram(
 input clk,
 input [7:0] addr,
 input [15:0] din,
```

```
          input we,
          output reg [15:0] dout
          );

         reg [15:0] ram_array [7:0];

      // Необязательная инициализация
      // initial
      // $readmemh("file_name", ram_array, <begin_addr>,
<end_addr>);

         always @(posedge clk)
             begin
             if (we) begin
               ram_array[addr] <= din;
             end
             dout = ram_array[addr];
             end

      endmodule
```

Графическое изображение модуля показано на рисунке 4.4.

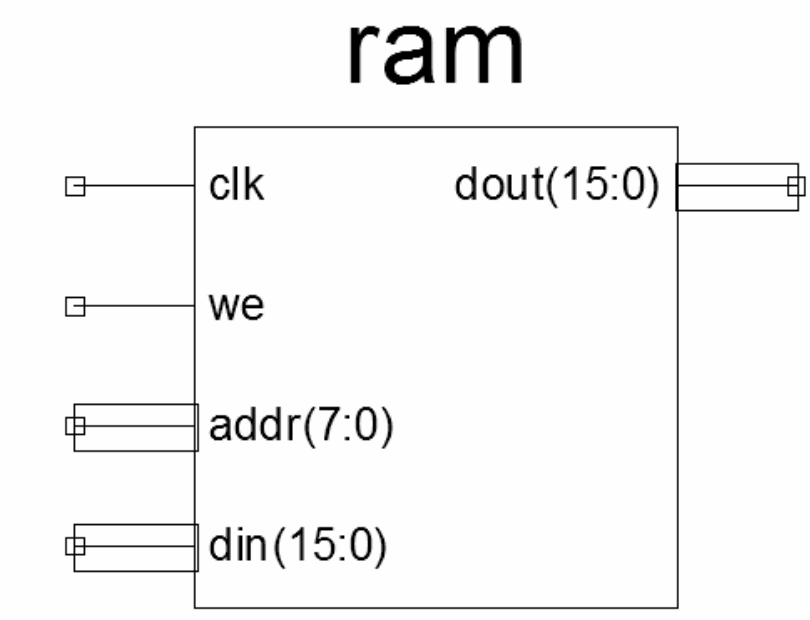


Рисунок 4.4 - Графическое изображение модуля ОЗУ с синхронным интерфейсом.

Из описания порядка работы модуля ОЗУ виден его недостаток, проявляющийся в том, что при записи в память невозможно одновременно читать ее содержимое. Например, при использовании блока памяти для хранения таблицы значений, непрерывно выдаваемых на цифро-аналоговый преобразователь, устройство отображения информации или на иное устройство, требующее непрерывного потока данных, возникнет проблема, связанная с тем, что для обновления содержимого памяти придется прервать процесс

чтения записанных в нее данных. От этого недостатка свободны многопортовые модули памяти, которые позволяют обращаться к одному и тому же массиву ячеек с помощью нескольких наборов линий `addr`, `din`, `we`, и имеют соответствующее количество выходных шин `dout`. Простейшим вариантом многопортовой памяти является двупортовая (dual-port memory), которая имеет достаточно много разновидностей. По функциональным возможностям второго порта двупортовая память подразделяется на simple dual-port, или pseudo dual-port («простая двупортовая», или «псевдо-двупортовая» память), и true dual-port («истинно двупортовая память»). Их отличием является то, что память true dual-port имеет два независимых и равноправных порта, по каждому из которых возможно проведение операций чтения и записи. У памяти simple dual-port один порт является универсальным (чтение и запись), а второй – только для чтения. Память такого типа вполне может быть использована в проектах, где требуется обеспечение непрерывного потока читаемых данных. В этом случае при необходимости перезаписи используется универсальный порт, а второй и используется для постоянного считывания. Графическое изображение simple dual-port памяти показано на рисунке 4.5.

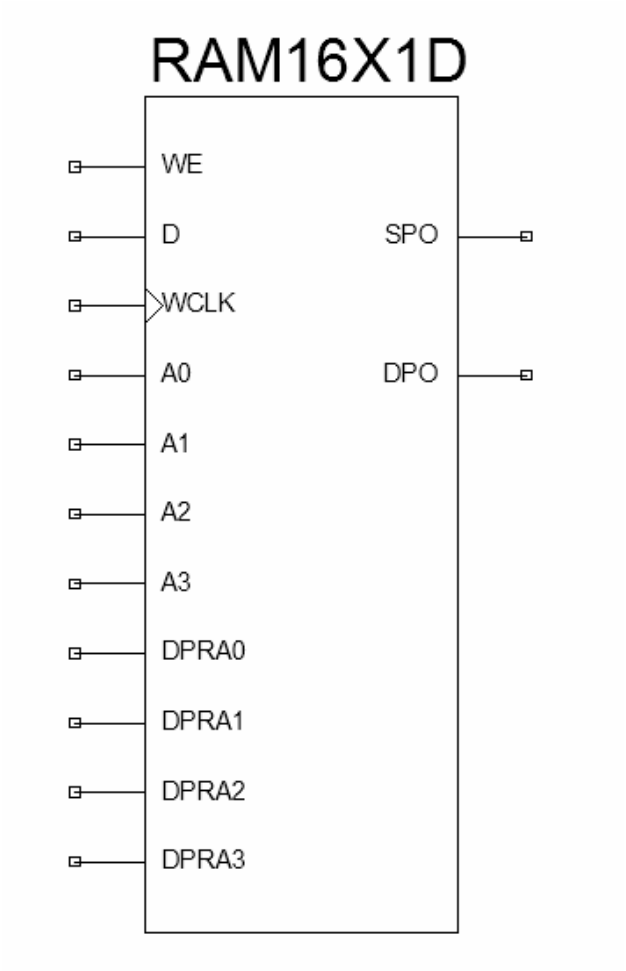


Рисунок 4.5 - Двупортовая память в конфигурации simple dual-port, построенная на базе логической ячейки FPGA

Логические генераторы программируемых ячеек FPGA представляют собой массивы статической памяти, хранящие таблицы истинности. Поэтому они могут быть использованы и в качестве модулей памяти, являясь при этом simple dual-port памятью. Такая память в терминологии FPGA называется также *распределенной* (*distributed*), поскольку распределена по программируемым ячейкам.

Блочная память, размещаемая в FPGA, является true dual-port. Графическое изображение аппаратного примитива, представляющего собой наиболее полный вариант интерфейса, показано на рисунке 4.6.

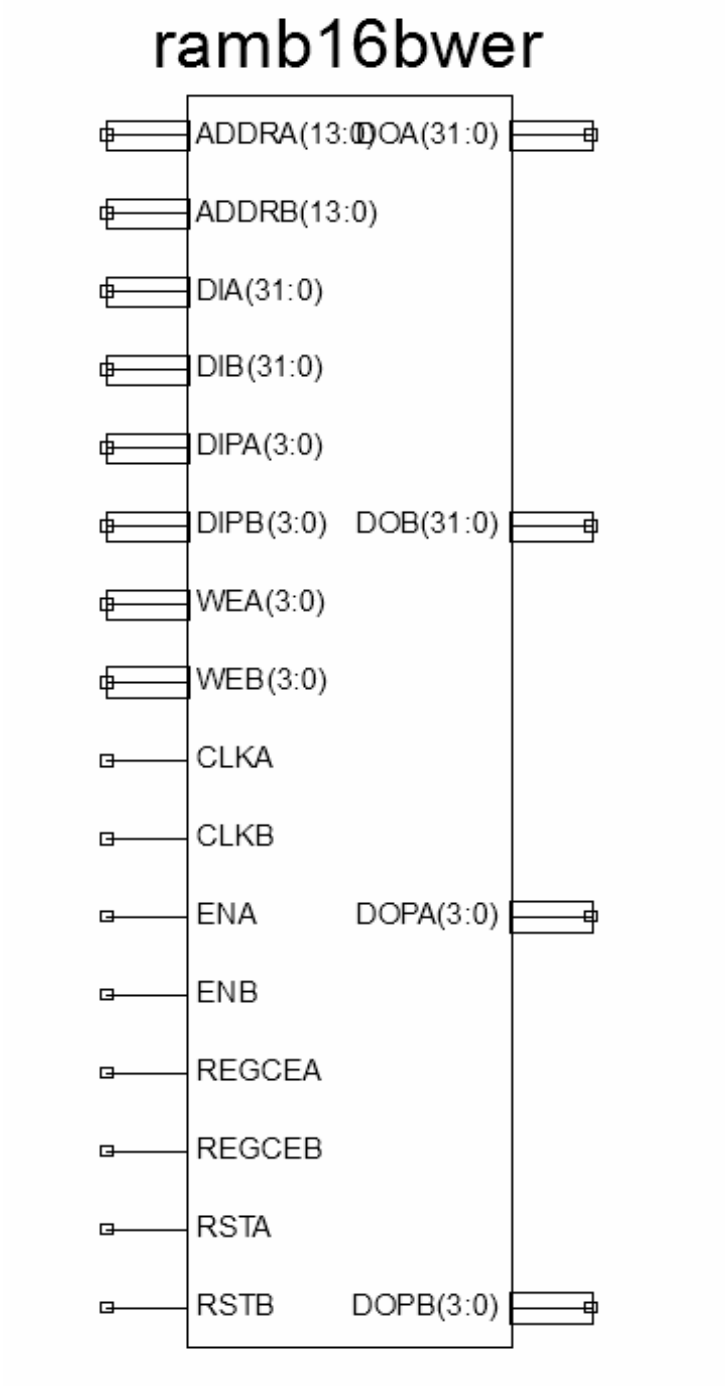


Рисунок 4.6 -Графическое изображение блока памяти.

Блок памяти имеет следующие порты:

- addra, addrb – адреса портов A и B соответственно;
- dia, dib – данные для записи для портов A и B (32 бита);
- dipa, dipb – дополнительные данные для записи (4 бита);
- wea, web – входы разрешения записи (побайтно);
- clka, clkb – тактовые сигналы для портов;
- ena, enb – входы разрешения работы блока памяти (при чтении состояние выходов не обновляется, если нет разрешающего сигнала);
- regcea, regceb – разрешение работы выходных регистров;
- rsta, rstb – сброс выходных регистров (не влияет на содержимое массивов памяти);
- doa, dob – выходы данных для портов A и B (32 бита);
- dopa, dopb – дополнительные выходы данных для портов A и B (4 бита).

Физически размещенные в FPGA блоки памяти являются 18-битными. Такая разрядность позволяет реализовывать схемы контроля четности, когда каждые 8 бит имеют дополнительный 9-й бит для хранения бита четности. Соответственно, каждые 16 бит имеют 2 дополнительных бита четности, а 32 – 4 бита. Для удобства работы с дополнительными битами в графическом представлении модуля они выделены в отдельные шины dipa, dipb, dopa, dopb.

Дополнительные биты не являются автоматически заполняемыми и представляют собой разряды, доступные для записи в них произвольных значений. Разработчик может выбирать требуемую ему разрядность, включая 9, 18 или 36 бит.

Блочная память является эффективным аппаратным ресурсом, который допускает работу на *системной тактовой частоте*. Это максимальная тактовая частота, на которой теоретически мог бы работать проект, если бы он не содержал цепей комбинаторной логики, проходящих более чем через один логический генератор, и слишком длинных трассировочных цепей. На практике работа на системной тактовой частоте оказывается доступной только для относительно компактных фрагментов проекта, и важно, что блочная память не ухудшает эти параметры. Следует также иметь в виду, что удельная стоимость блочной памяти ниже, чем памяти того же объема, реализованной на логических ячейках. Сложно указать точную границу объема, превышение которой делает блочную память однозначно более эффективным решением, однако можно ориентироваться на технические характеристики разных устройств (16x1 или 64x1 бит в логическом генераторе, 1024x18 бит в блоке памяти), а конкретный способ реализации памяти выбирать, исходя из доступных ресурсов проекта и требуемых технических характеристик.

Для принудительного использования заданного ресурса может потребоваться component instantiating, поскольку синтезаторы для ПЛИС при использовании автоматических настроек выбирают способ реализации памяти (блочная или распределенная), исходя из ее размера и режима работы (для true dual-port возможна реализация только в блочной памяти).

4.11 Контроллер UART

Аббревиатура UART обозначает Universal Asynchronous Receiver-Transmitter – «универсальный асинхронный приемопередатчик»

Интерфейс RS-232 является стандартным последовательным интерфейсом для

PC. Сопряжение устройства на базе ПЛИС с ЭВМ, выполненное без дополнительных внешних устройств (за исключением преобразователя уровней типа MAX232 или ADM232) предоставляет новые возможности как разработчику как этапе проектирования, так и пользователю созданного устройства. Кроме того, разработка контроллера RS-232 позволяет продемонстрировать довольно эффективный прием проектирования на VHDL – реализацию конечных автоматов для выполнения относительно сложных операций.

Для асинхронной передача по протоколу RS-232 достаточно всего двух сигнальных линий – TD (Transmit Data) и RD (Receive Data). Для электрического сопряжения сигналов потребуются стандартные устройства преобразования уровней RS-232 (лежащих в диапазоне –12…+12В) в логические уровни ТТЛ/КМОП. После преобразования к выводам ПЛИС окажутся подключены два сигнала, которые можно обозначит как *rx* и *tx*.

Протокол передачи по интерфейсу RS-232 представлен на рисунке 4.7. Передача начинается установкой низкого логического уровня на входе приемника. Низкий уровень сохраняется на время длительности одного бита, который называется стартовым. Далее передаются биты данных (число которых зависит от настроек протокола). В последовательных портах PC передача начинается с младшего бита. После передачи последнего бита данных вводится бит четности, формируемый по позитивной или негативной четности в зависимости от настроек протокола. Теми же настройками передача бита четности может быть отключена. В завершение передается стоповый бит, после которого линия остается в состоянии логической единицы вплоть до следующей посылки данных.

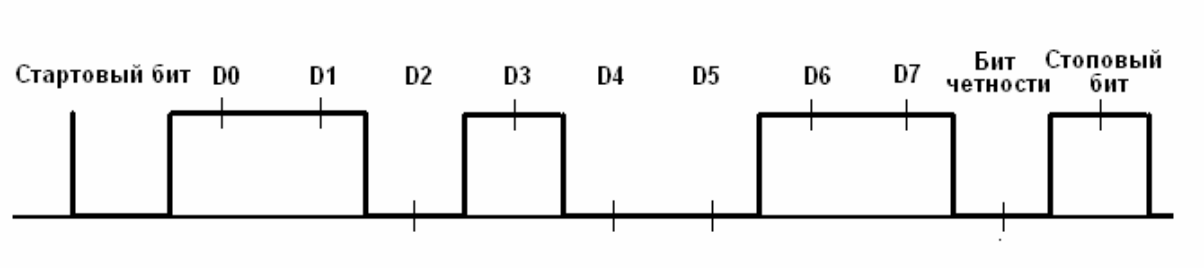


Рисунок 4.7 - Передача данных по протоколу RS-232.

Асинхронный протокол передачи накладывает достаточно жесткие требования на стабильность интервалов времени, в течение которых передаются отдельные биты. Для повышения надежности передачи желателен прием отдельных битов возможно ближе к середине этих интервалов. С той же целью можно использовать довольно эффективный прием многократного считывания состояния линии с последующим подсчетом количества принятых нулей и единиц.

При реализации приемника в ПЛИС необходимо в первую очередь решить проблему задания временных интервалов. Действительно, если длительность отдельных битов и может быть определена исходя из известных настроек протокола передачи, то ПЛИС, вообще говоря, не имеет собственного устройства отсчета времени. Следовательно, для синхронизации необходимо использовать внешний тактовый сигнал, причем его частота должна быть достаточно большой для обеспечения хорошего разрешения по времени.

Наличие внешнего тактового сигнала и асинхронных сигналов, управляющих состоянием приемника, позволяют использовать технологию проектирования конечных

автоматов. Под конечным автоматом понимается некое устройство, которое может находиться в фиксированном числе состояний, переходы между которыми совершаются при выполнении определенных условий. В ПЛИС внешний тактовый сигнал может быть эффективно использован для тактирования работы конечного автомата.

Рассмотрим временную диаграмму приема данных по последовательному интерфейсу (рисунок 4.8). Передача данных начинается со стартового бита, причем время ожидания в общем случае не определено. Разумным решением будет ввод в состав конечного автомата (далее КА) состояния *«ожидание начала»*. Из этого состояния его может вывести появление логического нуля на входе данных *rx*, что мы и отметим. После выхода из этого состояния необходимо подождать начала передачи младшего бита данных. В терминах КА это означает последовательный переход между вспомогательными состояниями. Эти переходы будут происходить с частотой следования внешних тактовых импульсов, и их количество (до середины младшего бита данных) определится длительностью стартового бита и половины длительности младшего бита данных.

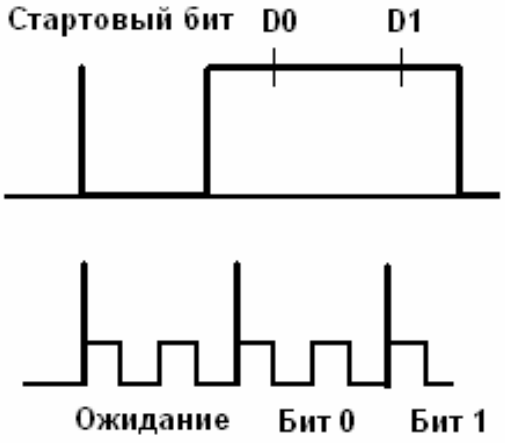


Рисунок 4.8 - Диаграммы состояний конечного автомата приемника RS-232.

Из рисунка видно, что длительности одного бита (стартового, стопового или данных) соответствует N тактов внешнего генератора. Следовательно, через N+N/2 тактов после начала стартового бита можно считывать младший бит данных, еще через N тактов – следующий бит и т.д. Всего до приема 8-го бита данных пройдет N + 7*N + N/2 тактов внешнего генератора. Поставим в соответствие каждому такту некоторое состояние конечного автомата и запишем последовательность переходов между состояниями.

```
Ожидание начала => Ожидание начала ИЛИ Стартовый бит
Стартовый бит => перейти к считыванию бита данных 0
Бит данных 0 => перейти к считыванию бита данных 1
...
Бит данных 7 => формированию сигнала «принято»
Принято => отключить сигнал «принято» и перейти к ожиданию начала
Остальные случаи => переход к следующему состоянию
```

При высокой частоте тактирования КА состояний оказывается довольно много. Однако ввиду простоты переходов между ними можно ввести переменную *st* (от state – состояние), которая будет хранить *номер* состояния КА. Пересчитав количество тактов (состояний) на один бит, можно получить номера состояний, в которых необходимо запоминать 0-й, 1-й, 2-й и т.д. биты данных.

4.12 Общие рекомендации по разработке проекта на базе FPGA

Достижение высокой производительности и надежности проекта на базе ПЛИС с архитектурой FPGA основывается на следующих подходах:

1. Широкое использование аппаратных ядер, таких как блочная память, аппаратные умножители и блоки «умножение с накоплением», сериализаторы/десериализаторы и др.
2. Использование синхронного подхода к описанию проектов, что хорошо соответствует архитектуре FPGA, выполняемых по технологическим нормам 90 нм и менее (Spartan-3, Virtex-4 и более поздние семейства).
3. Настройка параметров синтеза и трассировки в САПР с учетом желаемых характеристик проекта.

Вопросы использования аппаратных ядер были рассмотрены в предыдущих разделах. Следует особо следить за обязательным размещением в проекте генератора тактовых сигналов, не ограничиваясь глобальным тактовым буфером.

Для синхронного проекта наблюдаются следующие свойства:

- число тактовых цепей в проекте минимально, в идеальном варианте все модули используют один тактовый сигнал, формируемый аппаратным модулем FPGA (DCM, CMT, MMCM или иным, специфичным для данного семейства);
- все модули используют синхронный сброс;
- все модули используют только один перепад тактового сигнала (как правило, фронт);
- используются триггеры, а не защелки;
- при наличии нескольких тактовых сигналов, переход данных от одного тактового домена к другому осуществляется через специальные схемы синхронизации;
- выводы FPGA являются выходами регистра, входные сигналы записываются в регистры так быстро, как это возможно;
- критичные цепи конвейеризованы.

Приведенные рекомендации не являются исчерпывающими, однако позволяют рассчитывать на формирование хорошей основы для последующего получения надежно функционирующей схемы. Описания некоторых цифровых устройств на Verilog можно найти также в [4.1 - 4.3].

Литература к разделу 4:

4.1. Поляков А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры/ - М.: СОЛОН_Пресс, 2003. – 320 с.

4.2. Тарасов И. Е. Язык описания аппаратуры Verilog: Учебное пособие. – М.: МГТУ МИРЭА, 2011. – 132 с.

4.3. Стемпковский А.Л., Семенов М.Ю. Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: Учебное пособие. – М.: МИЭТ, 2005. -140 с.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ОСНОВАМ ПРОЕКТИРОВАНИЯ СИСТЕМ НА ПЛИС С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА VERILOG

Лабораторная работа №1. Основы проектирования цифровых устройств инструментами САПР ISE

Цель работы: освоить основные приемы работы со средой проектирования ISE13.2, получить навыки применения демонстрационной платы ALTYS для аппаратного тестирования.

Задание:

1. На основе микросхемы FPGA 6SLX45CS (семейство Spartan-6) в корпусе G324 разработать логическое устройство комбинаторной логики, реализующее функции 2НЕ-ИЛИ и 2И. При разработке устройства применить комбинацию встроенного библиотечного модуля 2И и модуля 2НЕ-ИЛИ на основе логического поведенческого описания.

2. В качестве макета для демонстрации разработки использовать плату ATLYS, содержащую комплект периферийных устройств для реализации проектов на основе микросхемы FPGA 6SLX45CSG324 семейства Spartan-6.

3. Входные сигналы формировать при помощи ползунковых выключателей, подключенных к входам ПЛИС.

4. Для тестирования разработанного устройства его выходы подключить к светодиодам, расположенным на демонстрационной плате.

Указания для выполнения работы №1:

Проектом в САПР ISE является совокупность файлов, которые содержат информацию, необходимую и достаточную для выполнения всех этапов разработки цифрового устройства.

При разработке цифровых устройств на базе ПЛИС Xilinx условно можно выделить следующие основные этапы проектирования:

- анализ задачи, разработка алгоритма работы устройства, разбиение проекта на модули, определение семейства ПЛИС, типа кристалла, корпуса, а также средств синтеза;
- разработка описания проектируемого устройства и его отдельных модулей в форме принципиальной схемы, кода поведенческого описания на языке HLD (Hardware Language Description);
- синтез модулей и всего устройства. Этап синтеза представляет собой процесс трансформации исходного HDL-описания проектируемого устройства в список цепей, элементы которого должны соответст133вовать архитектуре семейства ПЛИС, выбранного для реализации проекта;
- функциональное моделирование;
- размещение и трассировка проекта в кристалле;
- оптимизация устройства по временным характеристикам, потребляемой мощности и ресурсам ПЛИС;
- загрузка проекта в кристалл (программирование ПЛИС);
- подготовка технической документации проекта.

В начале устройство описывается в виде своей поведенческой (behavioral) моде-

ли, на которой отрабатывается задуманный алгоритм функционирования устройства. Затем эта модель вручную перерабатывается в синтезируемую модель, описанную на уровне регистровых передач (RTL-level). Такая модель, будучи оттранслированной компилятором-синтезатором, даёт проектную документацию в виде файла описания схемы устройства на уровне вентилей (EDIF-файл, Electronic Distribution International Format). При этом автоматически выполняется логическая оптимизация устройства. Одновременно этот файл автоматически преобразуется в HDL-модель на уровне вентилей.

Проект устройства в виде EDIF–файла принимается как исходный всеми САПР изготовления ПЛИС и СБИС. Эти САПР выполняют замену вентилей на библиотечные компоненты, их размещение на площади кристалла, трассировку межсоединений, проектирование масок, проверку соответствия проектным нормам и т.п. В результате записываются файлы проектной документации изготовления кристалла и его логической модели, учитывающей задержки как в вентилях, так и в межсоединениях. Эта модель также представляется на HDL.

Все этапы проектирования – алгоритмический, структурный, логический и технологический – сопровождаются моделированием устройства с помощью так называемого испытательного стенда (test fixture или testbench). Этот стенд представляет собой HDL-модель, составными частями которой являются модель тестируемого устройства и модели генератора тестовых сигналов и логического анализатора, проверяющих правильность функционирования. На всех этапах проектирования при верификации кода описания, проверки временных задержек после определения используемых стандартных библиотечных элементов, а также после трассировки и размещения внутри микросхемы ПЛИС (имплементации) может ис134пользоваться один и тот же испытательный стенд и те же тестовые файлы.

При выполнении работы рекомендуется руководствоваться указаниями по порядку выполнения проектных процедур в среде ISE 13.2, которые подробно изложены в разделе 2 настоящего пособия.

Лабораторная работа №2.
Иерархические структуры в языке Verilog

Цель работы: совершенствование навыков программирования Verilog для проектирования устройств комбинационной логики на основе ПЛИС в САПР ISE, закрепление навыков иерархического проектирования и разработки тестового покрытия, получение практических навыков самостоятельной работы с системами, содержащими ПЛИС.

Задание:

1. Используя разбиение на модули, на основе микросхемы FPGA 6SLX45CS (семейство Spartan-6) в корпусе G324 разработать логическое устройство комбинационной логики, преобразующие сигналы четырех входов в выходной сигнал в соответствии с рисунком 5.2.1 и таблицами 5.2.1 и 5.2.2 с вариантами заданий.

2. Выполнить моделирование работы разработанного устройства, обеспечивающее полное тестовое покрытие.

3. Составить план тестирования и провести аппаратный тест разработанного устройства, используя возможности демонстрационной платы ALTYS.

Указания для выполнения работы №2:

Если в проекте используются несколько одинаковых модулей, то после того, как синтезирован первый экземпляр и следует использовать функцию добавления в проект уже существующих компонентов: **Project – Add Source…** Для передачи сигналов между внутренними модулями использовать тип данных **wire**.

Для аппаратного тестирования использовать группы ползунковых переключателей и светодиодов платы ALTYS.

Таблица 5.2.1 – Задание функции модулей иерархического проекта.

1	**+**	сложение
2	**> >= < <=**	отношения
3	**&&**	логическое И
4	**||**	логическое ИЛИ
5	**==**	логическое равенство
6	**!=**	логическое неравенство
7	**&**	побитовое И
8	**|**	побитовое ИЛИ
9	**^**	побитовое исключающее ИЛИ
10	**^~**	побитовая эквивалентность

Таблица 5.2.2 – Варианты иерархического проекта.

Номера функций модулей из таблицы 2.2			
№ варианта	1-й модуль	2-й модуль	3-й модуль
1.	1	1	2
2.	2	2	3
3.	3	3	4
4.	4	4	5
5.	5	5	6
6.	6	6	7
7.	7	7	8
8.	8	8	9
9.	9	9	10
10.	10	10	1

Для тестирования устройства созданы два испытательных стенда ***uut1*** и ***uut2***, отличающиеся порядком подключения тестовых сигналов.

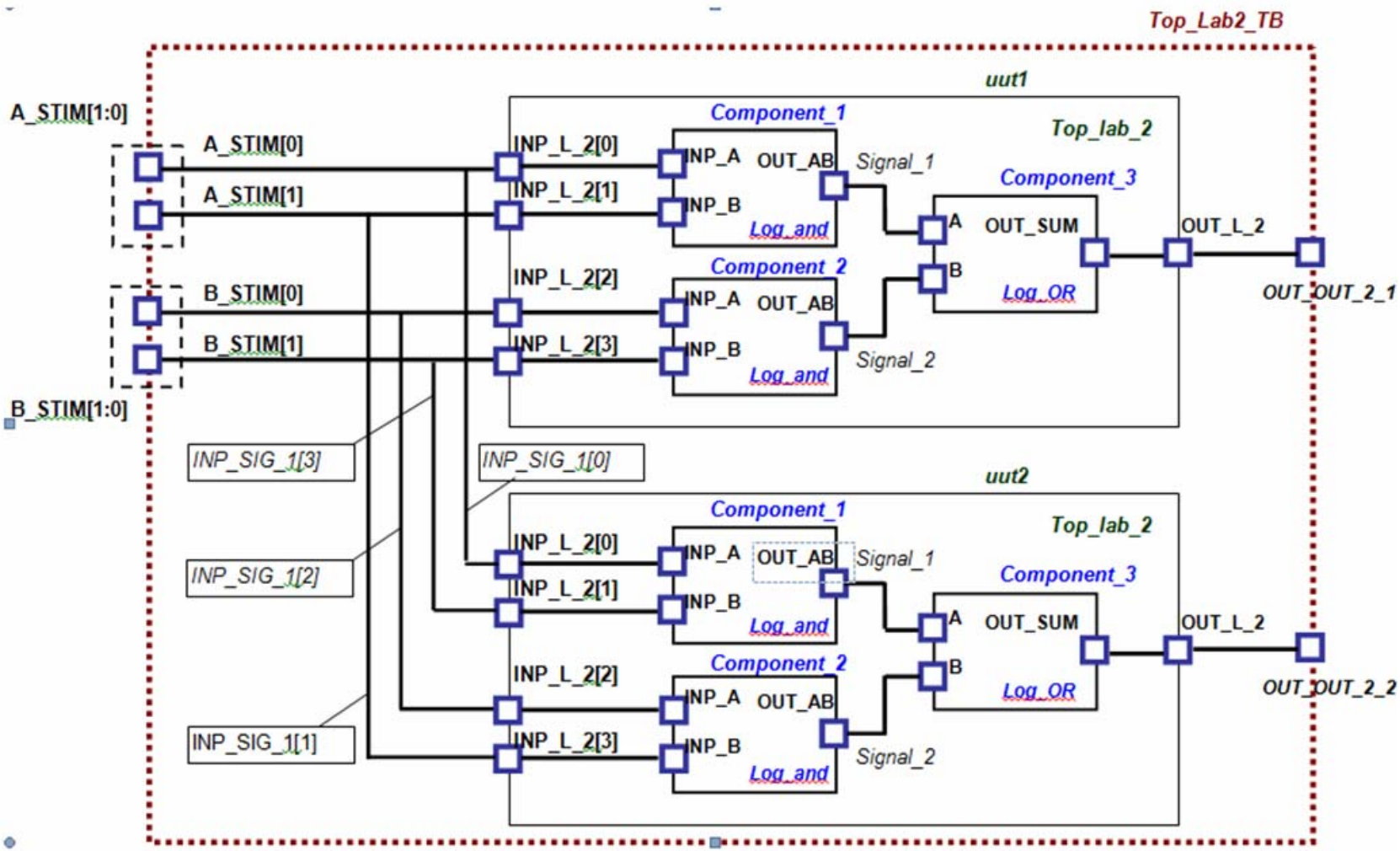


Рисунок 5.2.1 – Пример иерархического проекта: устройство ***Top_Lab_2***, включает два одинаковых модуля ***Log_and*** и один модуль ***Log_OR***.

Лабораторная работа №3. Двоичный счетчик

Цель работы: совершенствование навыков программирования на Verilog для проектирования устройств синхронной последовательностной логики на основе ПЛИС в САПР ISE, изучение способов использования управляющих сигналов синхронных схем на примере двоичного счетчика, закрепление навыков проектирования и разработки тестового покрытия.

Задание:

1. Используя конструкции языка Verilog, разработать синтезируемый в ПЛИС FPGA 6SLX45CSG324 двоичный счетчик с асинхронным сбросом и возможностями изменения направления счета и параллельной записи информации по фронту тактового импульса (аналог микросхем КР1533ИЕ10, ИЕ9, SN74ALS161AN). Разрядность счетчика должна задаваться изменяемым параметром verilog-кода.

2. Выполнить моделирование работы разработанного устройства, обеспечивающее полное тестовое покрытие.

Указания для выполнения работы №3:

1. Дополнительные справочные данные о микросхеме КР1533ИЕ10:

Символьное изображение четырехразрядного счетчика приведено на рисунке 5.3.1.

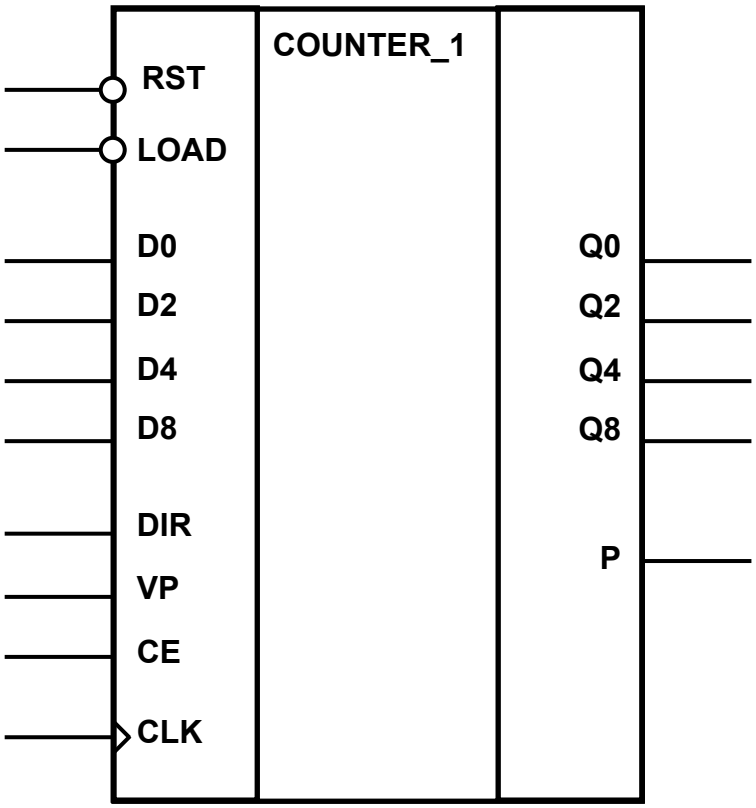


Рисунок 5.3.1 – Символьное изображение четырехразрядного счетчика на принципиальной схеме.

У микросхемы имеется четыре входа данных (D1, D2, D4 и D8), вход синхросигнала CLK, пять входов управления работой (RST, CE, VP, LOAD и DIR) выход переноса P и четыре выхода счета (Q1, Q2, Q4 и Q8).

За исключением сигнала RST, изменение состояния выходов счетчика происходит по фронту положительных импульсов, подаваемых на вход синхронизации CLK. Подача нуля на вход RST независимо от состояния других входов приводит к установке выходов микросхемы в состояние нуль.

Для обеспечения режима счета необходимо подать логическую единицу на вход RST, а также на входы разрешения параллельной записи LOAD и разрешения счета CE. Например, для обеспечения счета с числа, введенного при параллельной записи, логический нуль на входе LOAD должен быть изменен на логическую единицу одновременно с передним фронтом сигнала на входе CLK (переход из нуля в единицу) или при логическом нуле на входе CLK.

Подача логического нуля на вход CE останавливает счет, при этом выходы счетчика остаются в текущем положении счета.

При подаче логического нуля на вход LOAD микросхема переходит в режим параллельной записи информации с входов D1…DN. Запись происходит по передним фронтам импульсов синхросигнала CLK, т.е. для обеспечения параллельной записи логический нуль на вход LOAD и данные на входы D могут быть поданы как при логическом нуле, так и при логической единице на входе CLK, но обязательно должны удерживаться до момента перехода сигнала на входе CLK из логического нуля в логическую единицу (момент перезаписи данных).

Для обеспечения счета с числа, введенного при параллельной записи, логический нуль на входе LOAD должен быть изменен на логическую единицу или одновременно с переходом логического нуля в логическую единицу на входе CLK или при логическом нуле на входе CLK.

При подаче логического нуля на вход DIR осуществляется декрементный счет, при подаче логической единицы – счет инкрементный. Изменение направления счета происходит по фронту тактового импульса при условии, что входы RST и CE установлены в единицу.

На выходе переноса P импульс возникает в том случае, если на входе VP присутствует единица и на всех выходах счета Q установлены единицы. В остальных случаях на выходе P – логический нуль. Подача логического нуля на вход VP не запрещает счет, но запрещает выдачу сигнала логической единицы на выход P.

2. Цифровые схемы можно разделить на два больших класса: синхронные и асинхронные. В отличие от синхронных, в асинхронных схемах смена состояния может происходить в любой момент. Примером асинхронной схемы являются схемы, построенные с помощью базовых логических элементов при выполнении лабораторных работ №1 и №2. При проектировании цифровых устройств на базе ПЛИС рекомендуется использовать синхронный стиль проектирования. Синхронные схемы характеризуются наличием специального сигнала синхронизации (тактового сигнала), который определяет те моменты времени, по которым происходят изменения состояния схемы. Порядок использования синхронных сигналов сброса и разрешения счета можно продемонстрировать на примере двоичного счетчика. Это устройство, имеющее N разрядов, которые представляют число, линейно изменяющееся от 0 до 2N-1, после чего счет опять начинается с нуля. Подача сигнала «сброс» приводит к тому, что на следующем такте счетчик переходит в состояние 0. Снятие сигнала «разрешения счета» останавливает счетчик в достигнутом состоянии (не сбрасывая его).

3. Пример кода описания синхронного счетчика на языке Verilog (шаблон счетчика): приведен на рисунке 5.3.2. В данном примере используются три вложенных оператора if. Сначала выполняется проверка сигнала сброса счетчика в нулевое состояние. Если работа счетчика разрешена, т.е. сигнал сброса счетчика имеет нулевой уровень, то проверяется, разрешен ли доступ к счетчику или нет (первый вложенный оператор if). Если доступ к счетчику не разрешен (т.е. не выполняется CE == 1), то счетчик остановлен, и загрузить в него новое значение нельзя. Если условие выполнено, счетчик доступен, то проверяется какую операцию нужно выполнить: загрузку данных или счет. Если выбрана операция счета, то определяется, в каком направлении считать (третий оператор if): инкремент соответствует DIR == 1, декремент - DIR == 0. Соответственно, состояние счетчика, определяемое переменной Q увеличивается или уменьшается. Следует иметь в виду, что приведенный код является шаблоном-примером, для выполнения заданий по лабораторной работе №3 его нужно правильным образом отредактировать.

```
`timescale 1ns / 1ps

module Counter_1(RST, LOAD, DIR, CE, CLK, DIN, Q);
parameter NBRB = 8;
     input RST;
     input LOAD;
     input DIR;
     input CE;
     input VP,
     input CLK;
     input [NBRB-1:0] DIN;
     output P;
     reg P;
     output [NBRB-1:0] Q;
     reg [NBRB-1:0] Q;

always @ (posedge CLK or posedge RST)
     begin
          if (RST == 0) Q <= 'b0;
          else
               if (CE == 1)
                    begin
                         if (LOAD == 1) Q <= DIN;
                         else
                              begin
                                   if (DIR == 1) Q<=Q+1;
                                   else if (DIR == 0) Q<=Q-1;
                              end
                    end
     end

endmodule
```

Рисунок 5.3.2 – Шаблон кода Verilog-описания асинхронного двоичного счетчика.

4. Для задания тактовой частоты при моделировании работы счетчика использовать следующую конструкцию, задающую тактовый сигнал CLK с периодом 10 условных единиц времени:.

```
initial forever #5 CLK = ~CLK;
```

5. Для правильной организации временной диаграммы при моделировании работы счетчика рекомендуется составить таблицу событий моделирования по следующему образцу (см. таблицу 5.3.1):

Таблица 5.3.1 – План распределения по времени сигналов моделирования работы двоичного счетчика.

Текущее время или время выдержки сигнала в заданном состоянии, усл. ед.	Наименование сигнала	Характер изменения	Состояние счетчика
10	RST	переход в 0	сброс (обнуление)
20	RST	переход в из 0 в 1	разрешение счета
20	CE	установка в 1	разрешение счета с нулевого значения
…	…	…	…
150	DIR	переход из 1 в 0	изменение направления счета на обратное
…	…	…	…

Лабораторная работа №4. Делитель частоты

Цель работы: совершенствование навыков программирования на Verilog для проектирования устройств синхронной последовательностной логики на основе ПЛИС в САПР ISE, закрепление навыков проектирования и разработки тестового покрытия, совершенствование практических навыков самостоятельной работы с устройствами на основе ПЛИС и макетной демонстрационной платой ALTYS.

Задание:

1. Используя конструкции языка Verilog на основе счетчика, разработанного в работе №3 изготовить делитель частоты входного тактового в **n** раз (где целое число **n** – изменяемый параметр программы) и на основе микросхемы FPGA 6SLX45CS (семейство Spartan-6) в корпусе G324 разработать устройство, последовательно включающее и выключающее световые индикаторы платы ALTYS, формируя «бегущую строку».

2. Выполнить моделирование работы разработанного устройства, обеспечивающее полное тестовое покрытие.

3. Составить план тестирования и провести аппаратный тест устройства, исполь-

зуя возможности демонстрационной платы ALTYS. Наблюдая состояния светодиодных индикаторов, экспериментально подобрать коэффициент **n** деления тактовой частоты, таким образом, чтобы переключения выходных сигналов счетчика было удобно контролировать визуально.

Указания для выполнения работы №4:

При использовании синхронных схем часто возникает необходимость понизить тактовую частоту. На практике нижний предел частоты, подаваемой на вход ПЛИС, составляет 5, чаще 20 МГц, что может существенно превышать значение, необходимое для проекта. В то же время, для ПЛИС с архитектурой FPGA категорически не рекомендуется прерывать тактовый сигнал, использовать его для комбинирования с другими сигналами управления и т.д. Поэтому для понижения частоты необходимо использовать специальные приемы проектирования.

Для формирования сигнала с частотой, меньшей в N раз, можно использовать следующий прием. Описывается двоичный счетчик, значение которого изменяется в пределах 0 … N – 1. Далее с помощью оператора условного назначения выполняется следующая проверка:

```
assign led = (cnt == N-1) ? 1b'1 : 1b'0;
```

Оператор условного присваивания проверяет условие, записанное слева от знака «?» и назначает на выход значение, стоящее слева от «:», если условие выполняется, или значение, стоящее справа от «:», если условие не выполняется. В показанном примере логическая единица будет подана на выход, если значение счетчика ***cnt*** равно N-1, а это будет происходить один раз из N тактов. Таким образом, частота появления логической единицы на выходе ***led*** будет в N раз меньше, чем частота тактового сигнала.

Полученный таким образом сигнал не должен использоваться в качестве тактового для других частей схемы в ПЛИС. Вместо этого он используется в качестве сигнала разрешения счета.

Пример реализации устройства, в соответствии с требованиями задания выводящего сигналы на светодиоды, приведен на рисунке П.3.4.1.

Для аппаратного тестирования необходимо использовать генератор тактовых сигналов, установленный на плате ALTYS и подключенный **к выводу L15 FPGA** 6SLX45CSG324 (входной сигнал частотой 100 МГц), а также, при необходимости, группы ползунковых переключателей, светодиоды и кнопки платы ALTYS.

Поскольку частота тактового генератора слишком высока, для визуального наблюдения мигания светодиодов при аппаратном тестировании следует применить дополнительный модуль делителя частоты, представляющий собой такой же счетчик, выход которого подключен к основному счетчику как сигнал разрешения счета. Коэффициент деления задается параметром в коде и находится в пределах 1_000_000 - 10_000_000. Функциональная схема устройства приведена на рисунке 5.4.1.

Для запуска устройства, реализуемого сигналом ***EN***, используйте один из ползунковых переключателей или одну из кнопок на демонстрационной плате ALTYS.

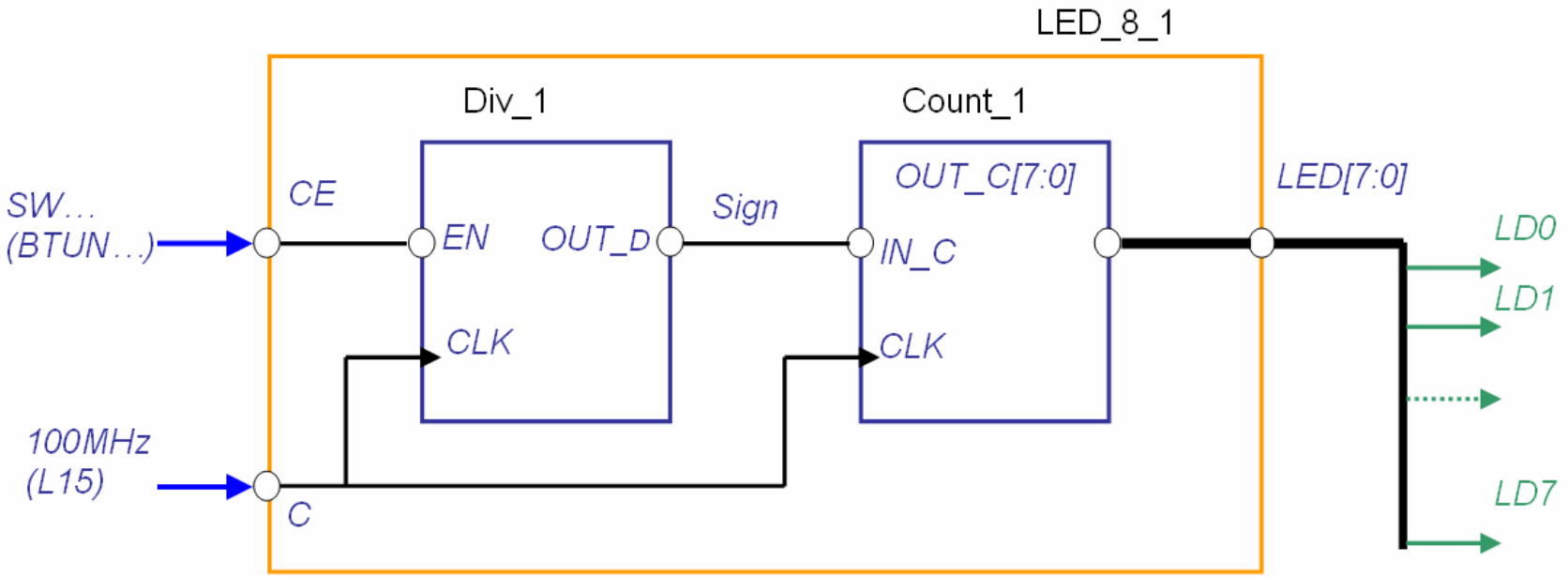


Рисунок 5.4.1 – Функциональная схема устройства вывода индикации счета на светодиоды платы ATLYS.

Таблица 5.4.1 – Варианты заданий:

№ варианта	Кнопка или переключатель разрешения счета	Кнопка изменения направления счета
1.	SW0	SW7
2.	BTUN0	SW0
3.	BTUN1	SW1
4.	BTUN2	SW2
5.	BTUN3	SW3
6.	SW4	SW7
7.	SW5	SW0
8.	SW6	SW1
9.	SW7	SW0

Лабораторная работа №5. Вычислитель

Цель работы: изучение типовой архитектуры вычислительных устройств, совершенствование навыков разработки и моделирования устройств и систем с помощью HDL-кода на примерах проектирования модуля памяти, конечного автомата, арифметико-логического устройства, компаратора, объединенных модулем верхнего уровня, представляющем собой простейшую вычислительную структуру.

Задание. Выполнить структурное описание для объекта вычислителя SIMPLE_CALC, находящегося на верхнем уровне иерархии и состоящего из следующих компонентов более низкого уровня:

- модуля памяти MEM (задание 5.1.),
- одного из компараторов COMP_BEH или COMP_RTL в соответствии с тем как используется HDL-описание: для моделирования или для синтеза (задание 5.2),
- модуль арифметико-логического устройства ALU (задание 5.3),
- модуль конечного автомата CNTRL_FSM (задание 5.4).
- разработать устройство «Вычислитель», включающее в приведенные выше модули (задание 5.5) и используя возможности демонстрационной платы ALTYS и провести аппаратное тестировании его работы.

Указания для выполнения работы №5:

Базовая структура ЭВМ определена фон Нейманом (1945 г.). Для того, чтобы ЭВМ была универсальным и эффективным устройством обработки информации, она должна строиться в соответствии со следующими принципами:

1. Информация кодируется в двоичной форме (специфика электронных схем –

простота и надежность работы) и разделяется на элементы информации, называемые словами. Слово обрабатывается в ЭВМ как одно целое и представляет собой машинный элемент информации.

2. Разнотипные слова информации хранятся в одной и той же памяти (?) и различаются по способу использования, но не по способу кодирования. Т.е., команды или данные выглядят совершенно одинаково, и только порядок использования слов в программе вносит различие в слова. В результате и сами команды могут модифицироваться и обрабатываться как числа.
3. Слова информации размещаются в ячейках памяти машины и идентифицируются номерами ячеек, называемыми адресами слов. Структурно память состоит из перенумерованных ячеек. Чтобы записать слово в память или выбрать слово из памяти необходимо указать адрес ячейки. Таким образом, адрес ячейки служит машинным идентификатором (именем) хранимой величины или команды. При этом выборка из памяти (чтение) не разрушает информацию, хранимую в ячейке.
4. Алгоритм представляется в форме последовательности управляющих слов, называемых командами, которые определяют наименование операции и слова информации, участвующие в операции (операнды). Алгоритм, представленный в терминах машинных команд, называется программой. В общем случае алгоритм в ЭВМ представляется в виде упорядоченной последовательности команд следующего вида:

Разряды полей	**bb…b**	**bb…b**	**bb…b**	**bb…b**	**bb…b**
Тип данных	**код операции**	**A1**	**A2**		**Ak**

Составные части команды называют полями. Вверху указаны номера разрядов полей. Здесь b – двоичная переменная, принимающая значения 0 или 1. Определенное число первых разрядов слова команды характеризует код операции (КОП). Последующие наборы двоичных переменных определяют адреса A1, A2…Ak операндов (аргументов и результатов), участвующих в операции, заданной своим кодом КОП.

5. Выполнение вычислений, предписанных алгоритмом, сводится к последовательному выполнению команд в порядке, однозначно определяемом программой. Первой выполняется команда, заданная пусковым адресом программы. Адрес следующей команды определяется в процессе выполнения текущей команды. Он может быть следующим по порядку (принцип следования) или адресом любой другой команды (переход). Процесс выполнения программы продолжается до тех пор, пока не будет подана команда остановки.

Следует подчеркнуть, что вычисления и все действия, проводимые машиной, определяются программой. Это позволяет изменить функции, выполняемые ЭВМ.

Перечисленные принципы функционирования ЭВМ предполагают наличие следующих устройств (базовая архитектура фон Неймана):

- арифметико-логическое устройство (АЛУ), выполняющее арифметические и логические операции;

- устройство управления (УУ), которое организует процесс выполнения программы;

- запоминающее устройство (ЗУ), или память для хранения программ и данных;

- устройство для ввода и вывода (ВУ) информации.

Архитектура ЭВМ определяет ее логическую организацию, состав и назначение функциональных устройств. Структура ЭВМ – совокупность элементов и связей между ними.

Обобщенная структура ЭВМ представлена на рисунке 5.5.1.

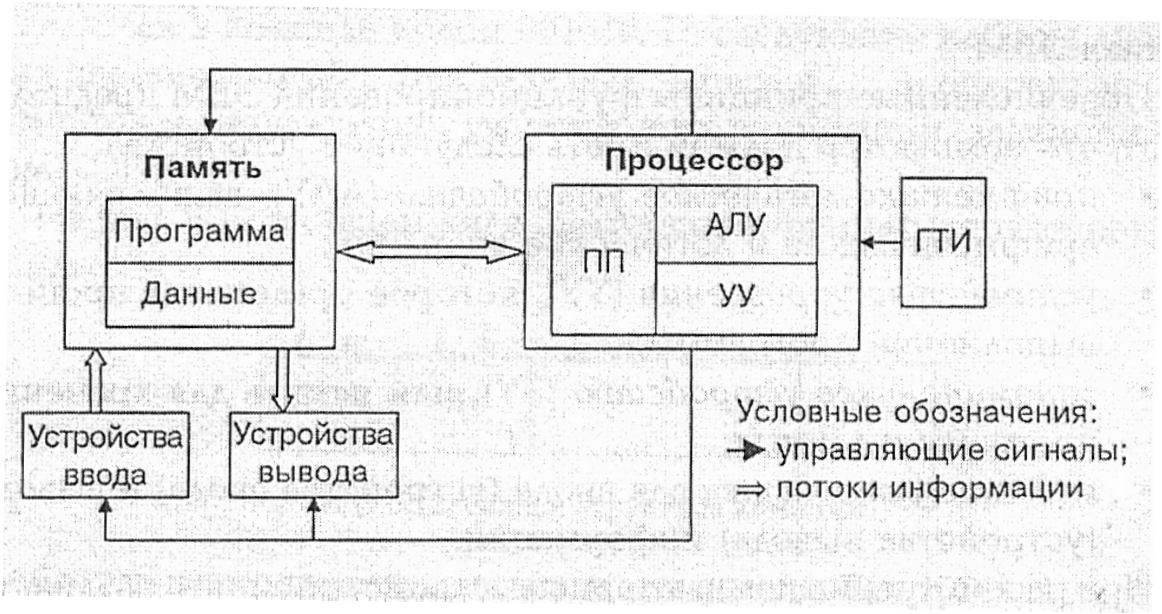


Рисунок 5.5.1 – Типовая архитектура ЭВМ.

ГТИ (генератор тактовых импульсов), осуществляет синхронизацию процессов передачи информации и выполнения команд (по фронтам тактовых импульсов). Промежуток времени между тактовыми импульсами определяет быстродействие ЭВМ.

УУ (устройство управления) и АЛУ (арифметико-логическое устройство) – объединены в процессор, предназначенный для обработки данных и управления работой ПК по заданной программе.

ПП (процессорная память) – специальные ячейки памяти называемые регистрами, которые располагаются внутри микропроцессора и служат для временного хранения информации.

В общем случае *регистры* – элементы памяти, каждый из которых может находиться в одном из двух устойчивых состояний (транзистор проводит или закрыт, конденсатор заряжен или разряжен, поляризованность в одном или другом направлении и т.п.). Регистр характеризуется числом битов информации, которые в нем могут храниться (разрядность). Пока включено питание, помещенная в нем информация остается до тех пор, пока она не будет заменена другой, процесс чтения информации из регистра не влияет на его содержимое и означает, что копия содержимого регистра будет создана и где-то сохранена. Каждый бит регистра передается по отдельному проводнику. Вся совокупность этих проводников образует шину.

Регистры, в частности, служат для ускорения работы микропроцессора и выполняют специальные функции:

- *аккумулятор,* в котором располагаются один из операндов или результат операции;

- *счетчик команд* – специальный регистр, содержимое которого увеличивается на единицу в момент выборки из памяти исполняемой команды и, если выбрана команда перехода, то оно может быть заменено на содержимое адресной части команды перехода; в конце цикла исполнения команды в этом счетчике всегда должен находиться адрес команды, которая выполняется за текущей (т.е. следующая по порядку команда или другая, к которой требуется перейти при выполнении условий, заданных кодом операции команды перехода);

- *регистр команд,* в котором размещается исполняемая команда;

- *регистр адреса*, содержащего адрес ячейки памяти, из которой будет считана команда, операнд или записан результат обработки;

- *регистр состояния*, специальный регистр, в котором хранятся признаки результата выполненной операции, по которым осуществляются переходы (передача управления).

Память ЭВМ – обеспечивает хранение команд и данных. Состоит из блоков одинакового размера (ячеек памяти), хранящих одно слово информации, все ячейки нумеруются, и адрес ячейки однозначно идентифицирует данные или команду в ней хранящуюся. При считывании содержимое ячейки не изменяется. При записи в ячейку хранимое в ней слово информации заменяется на новое.

Системный интерфейс – часть открытой архитектуры, набор цепей, связывающих процессор с памятью и контроллерами вычислительного устройства, алгоритм передачи сигналов по этим цепям, их электрические параметры и конструктивные элементы.

Исполнение команд процессором: определяется системой команд (микропрограмм), которые может распознавать и исполнять устройство управления (УУ). Выполняется принцип следования, согласно которому все команды выбираются из памяти последовательно. За адресом следующей команды следит специальный регистр – счетчик команд, в который УУ помещает адрес следующей команды. Обычно его содержимое увеличивается (инкрементируется) на одно или два слова памяти команд.

Если результат выполнения операции не пересылать в память, а сохранять в специальном регистре (аккумуляторе) то, можно построить систему команд, используя в каждой не более одного адреса. Команды в этом случае можно условно разделить на три группы:

- арифметические и логические (команды обработки), которые дают приказ на выполнение какой-либо арифметической или логической операции, используя в качестве операндов содержимое аккумулятора и содержимое адресуемой ячейки памяти (регистра данных);

- команды пересылки, которые дают приказ на обмен информацией между аккумулятором и памятью (через регистр данных), т.е. на загрузку аккумулятора содержимым адресуемой ячейки памяти или на запись в эту ячейку содержимого аккумулятора (результата обработки);

- команды установки битов слова состояния процессора и передачи управления, обеспечивающие переход к новой ячейке памяти с командой, которая должна быть выполнена следующей в программе при выполнении какого-либо условия или независимо от него.

Стандартная структура процессора для такой системы команд приведена на рисунке 5.5.2.

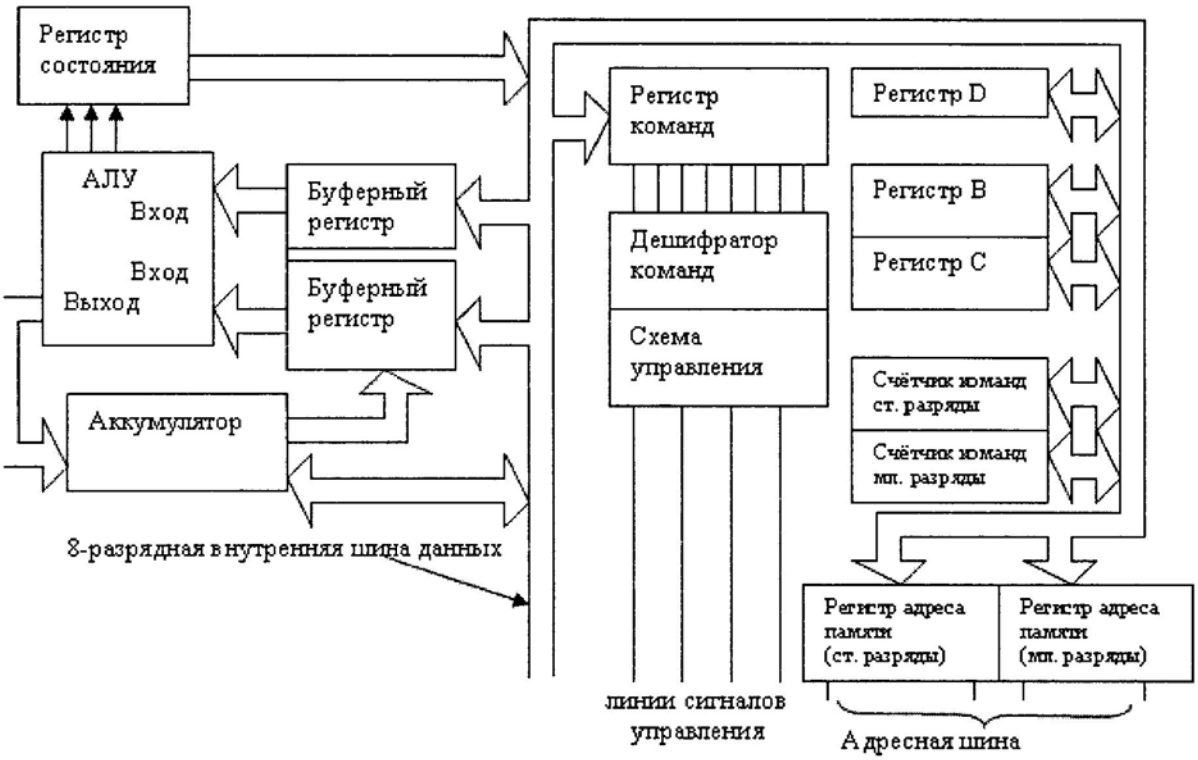


Рисунок 5.5.2 – Структурная схема микропроцессора.

Для решения задачи на такой ЭВМ следует:

1. Через устройство ввода информации загрузить в память ЭВМ программу решения задачи (алгоритм, написанный на языке ЭВМ, т.е. машинный код и исходные данные). Программа и исходные данные могут быть размещены в любой области памяти, начиная с ячейки с нулевым адресом или другим.
2. Сообщить процессору адрес ячейки памяти, в которой размещена первая (очередная) команда программы, для чего занести адрес этой ячейки в счетчик команд.
3. Каким либо способом заставить процессор приступить к исполнению программы (включить питание, нажать кнопку «Выполнить» или кнопку «Сброс»). В результате в память процессора поступит адрес первой (очередной) команды программы и произойдет пересылка содержимого этой ячейки адреса в регистр команд. С этого момента процессор начнет циклически выполнять стандартные операции выполнения программы.

Таким образом, работу вычислительной системы можно описать как циклическое повторение следующих операций:

- получить адрес памяти из счетчика команд PC;

- считать команду по данному адресу и поместить ее в регистр устройства управления;

- интерпретировать команду при необходимости считать из памяти значения операндов;

- по типу команды определить адрес следующей команды и соответствующим образом инкрементировать счетчик команд PC;

- выполнить команду в соответствии с существующим набором инструкций микропроцессора.

Структура простейшего вычислителя, являющегося предметом лабораторной работы №5, приведена на рисунке 5.5.3. Она аналогична типовой архитектуре вычислительной системы, приведенной выше на рисунке П5.3.2.

Вычислитель должен последовательно считать данные и коды операций из памяти, декодировать и выполнить команды над операндами, хранящимися в той же памяти и сравнить результат операции с эталонными значениями, также считанными из памяти. Выходным сигналом устройства является одноразрядный сигнал RESULT, который равен логической единице, в случае, если результат операции на выходе ALU совпадает с эталонным значением, считанным из памяти.

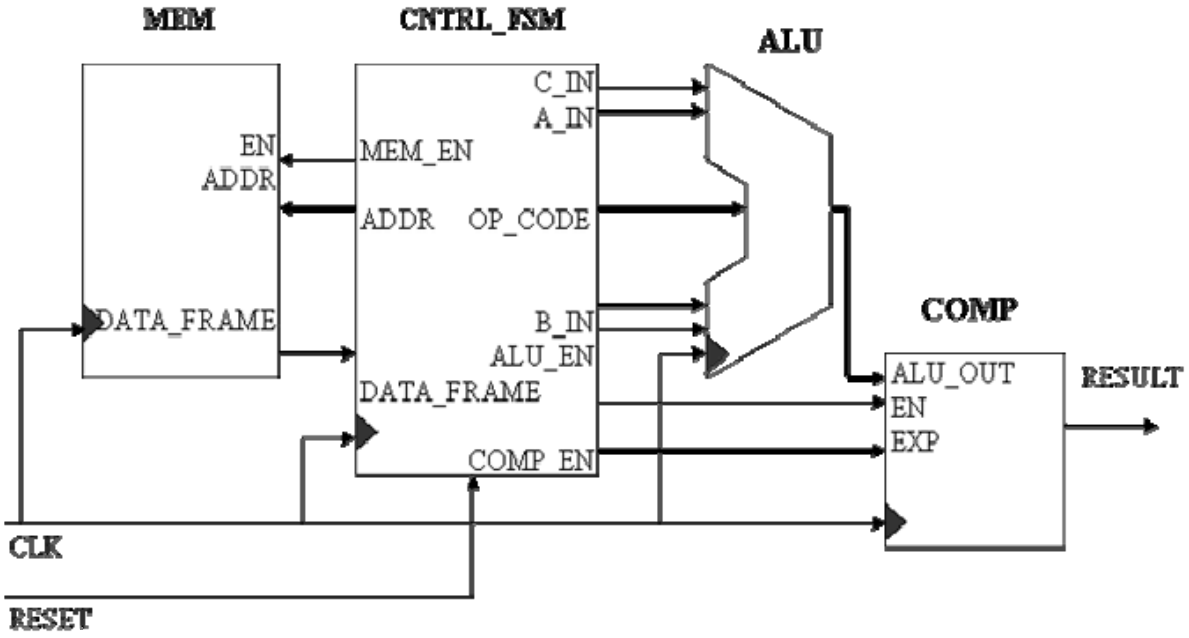


Рисунок 5.5.3 – Структурная схема устройства «Вычислитель».

Задание 5.1. Проектирование памяти

Напишите RTL-код для модуля памяти MEM с использованием файла, содержащего параметры настройки объема памяти. Выполните моделирование работы памяти, анализируя данные на выходе при чтении первых шести значений слов памяти.

Указания для выполнения задания 5.1:

Модуль синхронной памяти MEM является одним из компонентов проекта «Вычислитель» (SIMPLE_CALC). Параметры памяти: размер пространства адресов ADDR=3 и разрядность слова, хранящегося в одной ячейке WIDTH=17 удобно задавать во внешнем текстовом файле MY_HEADER. Эти параметры затем следует включить в

проект с помощью директивы компилятора **`*include “MY_HEADER.txt”***. Для создания текстового файла значений параметров из навигатора проекта используйте пункт меню ***New Source Wizard***, в котором выберете тип модуля ***User Document.*** После открытия шаблона модуля введите в него следующий текст:

```
` define WIDTH 17
` define ADDR 3
```

Затем сохраните файл и убедитесь, что в окне навигатора проекта во вкладке явился файл ***MY_HEADER.txt***.

Каждое слово в ПЗУ содержит данные (операнды) и команды для организации работы арифметико-логического устройства вычислителя. Формат слова данных и команд приведен в таблице 5.5.1:

Таблица 5.5.1 – Формат 17-ти разрядного слова памяти модуля MEM.

Назначение данных из памяти	Первый операнд	Второй операнд	Бит переноса	Код операции	Эталонное значение результата
Обозначение на структурной схеме	A_IN	B_IN	C_IN	OP_CODE	EXP
Разряды слова	16…13	12…9	8	7…4	3…0

Добавьте в проект новый файл с кодом ПЗУ ***MEM.v*** и отредактируйте шаблон модуля таким образом, чтобы входной сигнал ***EN*** =1 разрешал считывание данных из памяти, синхронизированное сигналом ***CLK***. Применяя операторы **always, if** и **case**, напишите код, реализующий выдачу данных на 17-ти разрядный выход DATA_FRAME в соответствии с таблицей 5.5.2. При выборе любого другого адреса ПЗУ, считать, что значения данных не определены (17’bx).

Таблица 5.5.2 – Данные памяти, передающиеся на выход DATA_FRAME в соответствии с заданным адресом.

Адрес	Данные	Десятеричное значение
3’b000	17’b0001_1000_1_0000_0010	12546
3’b001	17’b1001_0100_0_0001_1101	75805
3’b010	17’b0101_1010_1_0001_0000	46352
3’b011	17’b0001_0010_0_0010_1110	9262
3’b100	17’b0011_0001_1_0010_0010	25378
3’b101	17’b0000_1001_0_0111_1111	4735

Для моделирования работы ПЗУ примените конструкцию **initial**, внутри которой через определённые промежутки времени задавайте адреса ПЗУ и контролируйте значение данных на выходе памяти при каждом фронте сигнала CLK. Для анализа данных используйте представление DATA_FRAME в десятичном формате. Для этого выделите

этот сигнал в окне симулятора и в контекстном меню в выпадающей вкладке выберите пункты **Radix - Unsigned Decimal**.

Задание 5.2. Проектирование компаратора

Используя оператор ***if/else***, разработайте коды двух версий описаний для модуля **COMP** (синхронный компаратор). В первом случае поведенческое описание должно быть предназначено для испытательного стенда, выдающего сообщение об ошибке с помощью директивы ***$dislplay***. Во втором случае – напишите только синтезируемый RTL код, также используя конструкцию ***if/else***. Проверьте работу компаратора с помощью испытательного стенда.

Указания для выполнения задания 5.2:

Компараторы применяются для сравнения двух входных кодов и выдачи на выходы сигналов о результатах этого сравнения. Модуль компаратор COMP является одним из компонентов проекта «Вычислитель» (SIMPLE_CALC). В окончательном варианте в код SIMPLE_CALC модуль компаратора должен быть подключен при помощи процедуры условного подключения модуля в зависимости от того, решается ли задача синтеза всего устройства или задача тестирования его работоспособности.

Используйте разрядности и обозначения для портов компаратора, приведенные в таблице 5.5.3.

Таблица 5.5.3 – Разрядности и обозначения для портов модуля компаратора COMP.

Имя порта	Разрядность, бит	Описание
COMP_EN	1	вход, цепь разрешения работы компаратора, генерируется конечным автоматом
EXP	4	вход, эталонное значение, генерируется выходным сигналом памяти
ALU_OUT	4	вход, результат операции ALU
RESULT	1	выход, результат сравнения выходного сигнала ALU с эталонным значением.
CLK	1	вход синхронизации

Для выдачи сообщения об отрицательных результатах моделирования запишите следующую строку кода:

```
$display ($time, "_Simulation mismatch occurred, ALU_OUT is not equal to EXP");
```

согласно которой в момент времени **time** на экран монитора выдается сообщение «Обнаружено несоответствие: выходной сигнал ALU_COMP не совпадает с значением EXP».

Используя процедуру **View RTL Schematic,** изучите, как разработанный код синтезирован в схему. Обратите внимание, что сообщение об ошибке не синтезируется.

Для проверки функционирования модуля COMP последовательно задайте два варианта выходных сигналов ALU_OUT и убедитесь, что в случае несовпадения его значения с EXP на монитор выдается соответствующее сообщение.

Задание 5.3. Проектирование АЛУ

Напишите RTL-код для описания арифметико-логического устройства ALU, представляющего собой один из модулей устройства «Вычислитель» (SIMPLE_CALC). С помощью испытательного стенда проверьте работоспособность разработанного описания.

Указания для выполнения задания 5.3:

Арифметико-логическое устройство (АЛУ) предназначено для выполнения логических и арифметико-логических операций над операндами. Как правило, АЛУ является составной частью интегральной схемы микропроцессора, и перечень функций АЛУ определяется набором команд конкретного типа микропроцессора. Пример АЛУ, представляющего собой отдельную микросхему – четырехразряное скоростное АЛУ К155ИП3.

В проекте вычислителя Verilog-код модуля АЛУ должен описывать четырёхразрядные входы операндов A и B, четырёхразрядный вход выбора (кода) операций OP_CODE, вход переноса C_IN, вход разрешения выполнения операций EN и вход сигнала синхронизации CLK. Результат операции вырабатывается на выходе Y (рисунок 5.5.4).

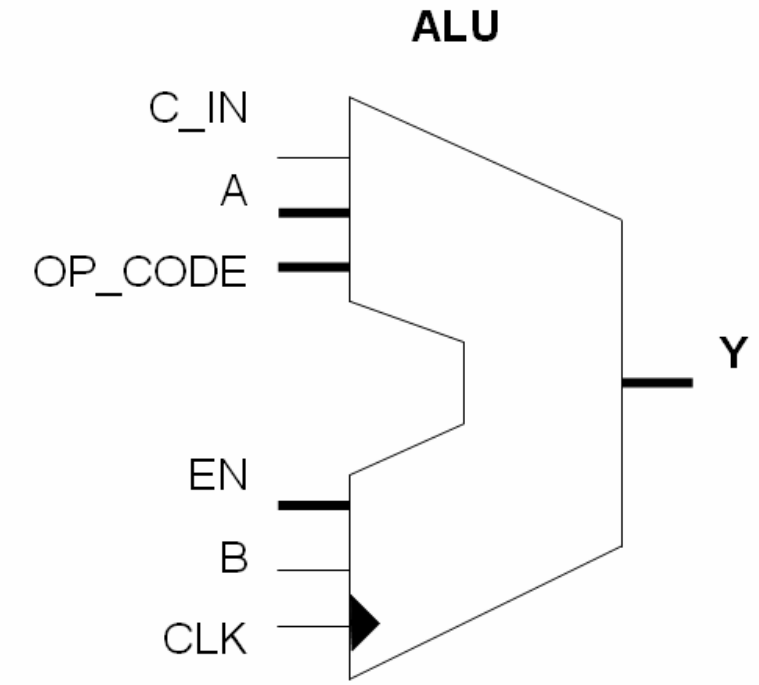


Рисунок 5.5.4 – Входные и выходные сигналы модуля ALU.

При написании кода для выбора конкретной операции используйте таблицу 5.5.4 и оператор **case**, аргументами которого служит результат конкатенации значений кода

операции OP_CODE и бита переноса C_IN:

```
assign OP_CODE_CI = {OP_CODE,C_IN};
```

Таблица 5.5.4 – Функции АЛУ в зависимости от состояния входов OP_CODE и C_IN.

OP_CODE	C_IN	Операция	Примечание
0000	0	Y = A	-
0000	1	Y = A +1	Инкремент
0001	0	Y = A + B	Сложение
0001	1	Y = A + B + 1	Сложение с переносом
0010	0	Y = A + (~B)	Сложение с инверсией
0010	1	Y = A + (~B) + 1	Вычитание
0011	0	Y = A – 1	Декремент
0011	1	Y = B	-
0100	0	Y = A & B	И
0101	0	Y = A | B	ИЛИ
0110	0	Y = A ^ B	ИСКЛЮЧАЮЩЕЕ ИЛИ
0111	0	Y = ~ A	Инверсия
1000	0	Y = 0	Установка нуля

Для проверки функционирования разработанного модуля составьте проверочную таблицу (например, аналогичную, таблице 5.5.5), в которой для нескольких значений пар входных операндов A и B, при заданных кодах операций вычислите ожидаемые значения.

Для задания входных сигналов при моделировании используйте шаблон испытательного стенда, в котором входное воздействие передается 5-ти разрядной цепью OP_CODE_CI_SIG, разделяющейся затем на две цепи OP_CODE и C_IN в соответствии с примером, приведенным на рисунке 5.5.5.

Занесите результаты, полученные в ходе моделирования, в таблицу и сравните их с вычисленными значениями сигнала Y.

Таблица 5.5.5 – Ожидаемые и полученные в ходе моделирования результаты работы АЛУ (тестовая таблица).

A (операнд 1)	B (операнд 2)	Операция	OP_CODE	C_IN	Y	Ожидаемый результат
0001	1000	Y = A	0000	0		**0001**
0001	1000	Y = A +1	0000	1		**0010**
0001	1000	Y = A + B	0000	0		**1001**
…	…	…	…	…	…	…
0001	1000		1000	0		**0000**

```
`timescale 1ns / 1ps

module ALU_TB_v;

  wire[3:0] OP_CODE;
  reg [3:0] A = 4'b0001;
  reg [3:0] B = 4'b1000;
  wire C_IN;
  reg CLK = 1'b0;
  reg EN = 1'b1;

  wire [3:0] Y;

  reg [4:0] OP_CODE_CI_SIG ;

    assign  OP_CODE = OP_CODE_CI_SIG[4:1];
    assign  C_IN = OP_CODE_CI_SIG[0];

  ALU uut (
        .CLK(CLK),
        .OP_CODE(OP_CODE),
        .A(A),
        .B(B),
        .C_IN(C_IN),
        .EN(EN),
        .Y(Y));

always #10 CLK = ~CLK ;

initial
begin
  #100 OP_CODE_CI_SIG = 5'b00000;
  #100 OP_CODE_CI_SIG = 5'b00001;
  #100 OP_CODE_CI_SIG = 5'b00010;
  #100 OP_CODE_CI_SIG = 5'b00011;
  #100 OP_CODE_CI_SIG = 5'b00100;
  #100 OP_CODE_CI_SIG = 5'b00101;
  #100 OP_CODE_CI_SIG = 5'b00110;
  #100 OP_CODE_CI_SIG = 5'b00111;
  #100 OP_CODE_CI_SIG = 5'b01000;
  #100 OP_CODE_CI_SIG = 5'b01010;
  #100 OP_CODE_CI_SIG = 5'b01100;
  #100 OP_CODE_CI_SIG = 5'b01110;
  #100 OP_CODE_CI_SIG = 5'b10000;
end

endmodule
```

Рисунок 5.5.5 – Пример кода тестового файла для моделирования ALU.

Задание 5.4. Проектирование конечного автомата

Напишите RTL-код, реализующий конечный автомат устройства «Вычислитель» (SIMPLE_CALC). С помощью испытательного стенда проверьте работоспособность разработанного описания.

Указания для выполнения задания 5.4:

Конечный автомат CNTRL_FSM является главным компонентом проекта «Вычислитель» (SIMPLE_CALC) поскольку он управляет работой остальных модулей устройства. В данном случае для организации машинного цикла используется автомат Мура, в котором выходные сигналы зависят только от текущего состояния автомата. Описание работы конечного автомата в виде схемы графов, отражающих все его состояния, приведено на рисунке 5.5.6.

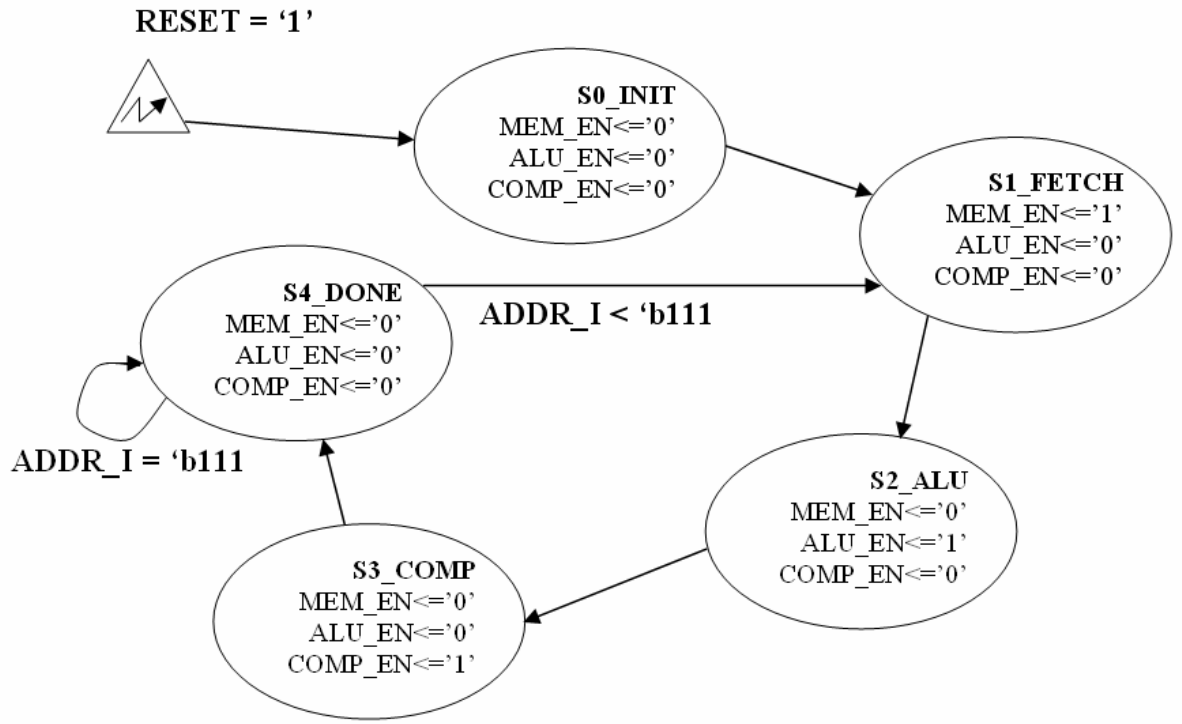


Рисунок 5.5.6 – Состояния и переходы конечного автомата, организующего работы устройства «Вычислитель».

Назначение конечного автомата – последовательное автоматическое выполнение всех операций внутреннего цикла работы вычислительной системы. В данном случае внутренний цикл вычислительной системы состоит в следующем: 1) начальное состояние; 2) считывание слова памяти, и выделение из считанного слова данных значений операндов, кода операции, значения бита переноса и эталонного значения результата; 3) выдача команды ALU выполнить ту или другую операцию в соответствии с ее кодом; 4) выдача компаратору команды разрешения сравнить полученный в АЛУ результат с эталонным значением и вывести результат сравнения; 5) определение следующего адреса для считывания слова из памяти и переход в состояние ожидания **`S4_DONE`** или

в состояние начала нового внутреннего цикла **S1_FETCH**. Из состояние ожидания автомат может выйти только после получения сигнала RESET. Все переходы из одного состояния в другое осуществляются синхронно по переднему фронту сигнала CLK.

По условию задания команды и данные для работы устройства содержатся в семи первых словах памяти. Таким образом, автомат должен последовательно считать семь слов памяти, выделить во внутреннем цикле из считанных данных операнды, коды операций, бит переноса и эталонное значение результата, дать команду ALU выполнить операцию, затем разрешить компаратору сравнить полученный результат с эталонным значением.

Рекомендуемая последовательность действий при программировании описания конечного автомата из проекта «вычислитель» приведена ниже.

Создайте в новом проекте файл с описанием конечного автомата ***CNTRL_FSM.v*** и определите для него входные и выходные порты в соответствии с рисунком 5.5.3:

```
`timescale 1ns / 1ps
module CNTRL_FSM(
     input CLK, RESET,
     input [16:0] DATA_FRAME,
     output reg [3:0] A_IN, B_IN, OP_CODE, EXP,
     output reg [2:0] ADDR,
     output reg COMP_EN, ALU_EN, C_IN, MEM_EN
     );
```

Задайте 5-ти разрядные локальные параметры для каждого состояния автомата в соответствии с рисунком 5.5.6:

```
localparam [4:0]
     S0_INIT   = 5'b00001,
     S1_FETCH  = 5'b00010,
     S2_ALU    = 5'b00100,
     S3_COMP   = 5'b01000,
     S4_DONE   = 5'b10000;
```

Объявите пятиразрядные переменные CURR_STATE и NEXT_STATE (тип ***reg***), предназначенные для хранения значений текущего состояния автомата:

```
reg [4:0] CURR_STATE, NEXT_STATE;
```

Объявите трехразрядную переменную ADDR_I (тип ***reg***), играющую в вычислителе роль внутреннего инкрементного счетчика машинных циклов и предназначенную для хранения определения момента перехода автомата в режим ожидания (состояние **S4_DONE)**:

```
reg [2:0] ADDR_I;
```

Запишите фрагмент кода синхронизации всех внутренних событий сигналом CLK, и то, что сигнал RESET, установленный в единицу активизирует асинхронный сброс автомата в начальное состояние:

```
always @ ( posedge CLK, posedge RESET)
     if (RESET == 1'b1)
          begin
          CURR_STATE <= SO_INIT;
          ADDR <= 3'b000;
          end
     else
          begin
```

```
CURR_STATE <= NEXT_STATE;
ADDR <= ADDR_I;
end
```

При написании кода примите во внимание следующие особенности входных и выходных сигналов:

DATA_FRAME – 17-ти битный вектор, считываемый из памяти и содержащий всю информацию для выполнения одного машинного цикла в соответствии с таблицей 5.5.1. Для установки значений сигналов используйте конструкцию **always @ *** и переназначение индексов:

```
always @ *
     begin
     A_IN      = DATA_FRAME[16:13] ;
     B_IN      = DATA_FRAME[12:9] ;
     C_IN      = DATA_FRAME[8] ;
     OP_CODE   = DATA_FRAME[7:4];
     EXP       = DATA_FRAME[3:0] ;
```

Сокращенная конструкция «**always @ *** » означает, что в списке чувствительности присутствуют все входные сигналы, которые могут повлиять на значения выходов. Эта конструкция используется в том случае, когда требуется описать элементы, которые изменяются, как только изменяются значения одного из их входов, и только для описания блоков содержащих комбинаторную логику или логические вентили. Внутри этой конструкции допускается использование только блокирующих (=) присваиваний.

ADDR – выходной сигнал, который должен инкрементироваться с каждым выполнением внутреннего цикла автомата для организации обращения к следующей ячейки памяти и считывания следующего слова с операндами и кодом операции. Адрес памяти также используется внутри цикла как условие для определения момента остановки работы конечного автомата. Поэтому в коде следует использовать дополнительную переменную - внутренний адрес **ADDR_I**, также увеличивающийся на единицу после завершения каждого полного внутреннего цикла работы автомата. Следует принять меры, чтобы предотвратить возможность защелкивания внутреннего адреса при неопределенности текущего состояния автомата. Для этого одновременно с инициализацией входных сигналов, следует задать также конкретное значение внутреннего адреса, равное текущему внешнему адресу:

```
ADDR_I = ADDR;
```

Для переходов конечного автомата из одного состояния в другое целесообразно использовать оператор case, условием для которого служит его текущее состояние:

```
case (CURR_STATE)

     S0_INIT:
     begin
          MEM_EN  = 1'b0;
          ALU_EN  = 1'b0;
          COMP_EN = 1'b0;

          NEXT_STATE = S1_FETCH ;
     end
```

```
      S1_FETCH:
      begin
           MEM_EN  = 1'b1;
           ALU_EN  = 1'b0;
           COMP_EN = 1'b0;

           NEXT_STATE = S2_ALU ;
      end

     S2_ALU:
      begin
           MEM_EN  = 1'b0;
           ALU_EN  = 1'b1;
           COMP_EN = 1'b0;

           NEXT_STATE = S3_COMP ;
      end

      S3_COMP:
      begin
           MEM_EN  = 1'b0;
           ALU_EN  = 1'b0;
           COMP_EN = 1'b1;

           NEXT_STATE = S4_DONE ;
      end

      S4_DONE:
      begin
           MEM_EN  = 1'b0;
           ALU_EN  = 1'b0;
           COMP_EN = 1'b0;
           if ( ADDR == 3'b111 )
                begin
                     NEXT_STATE = S4_DONE;
                end
           else
                begin
                     NEXT_STATE = S1_FETCH ;
                     ADDR_I = ADDR + 1;
                end
      end
 default:
      begin
           NEXT_STATE = S0_INIT;
           MEM_EN = 1'b0;
           COMP_EN = 1'b0;
           ALU_EN =  1'b0;
```

```
        end
    endcase
end             // относится к always @ *
endmodule       // относится к module CNTRL_FSM
```

При написании кода испытательного стенда ***CNTRL_FSM_TB.v*** задайте входные воздействия так, чтобы проверить следующие события:

- по сигналу **RESET**==1, автомат приходит в свое начальное состояние;
- сигнал **ADDR** должным образом инкрементируется (каждый раз за три периода сигнала **CLK**);
- сигналы **MEM_EN, ALU_EN** и **COMP_EN** выдаются на выход в нужные моменты времени);
- переходит ли FSM в состояние **S4_DONE**, если сигнал **ADDR_I** достигает своего предельного значения?

Для проверки функционирования задайте периодический тактовый сигнал с частотой 50 МГц (**`forever #10 CLK = ~CLK)`** и изменяйте через определенные промежутки времени значения слова данных **DATA_FRAME,** также как это было выполнено при выполнении заданий 5.1 и 5.3. Пример кода испытательного стенда:

```
initial
  begin
       // Initialize Inputs
       CLK = 0;
       RESET = 0;
       DATA_FRAME = 0;
       forever #10 CLK = ~CLK ;
  end

initial
  begin
       #50 RESET = 1'b1;
       #25 RESET = 1'b0;
  end

initial
  begin
       #1    DATA_FRAME = 17'b1100_0011_1_0000_1111 ;
       #100  DATA_FRAME = 17'b0011_0011_1_1110_1100 ;
       #100  DATA_FRAME = 17'b1101_1011_1_1000_1111 ;
       #100  ...
  end
endmodule // это относится к module CNTRL FSM TB
```

При анализе результатов моделирования учтите, что сигналы модулей низкого уровня **UUT** не отображаются в окне временной диаграммы автоматически. Если требуется отобразить сигналы модулей нижних уровней иерархии, следует найти сигналы соответствующего компонента **UUT** в иерархическом списке сигналов (вкладка **Instants and Process Name)** и назначить нужные сигналы для вывода принудительно в

ручном режиме, поместив соответствующие имена из окна **Objects** в окно отображаемых сигналов **Name** (пример приведен на рисунке 5.5.7).

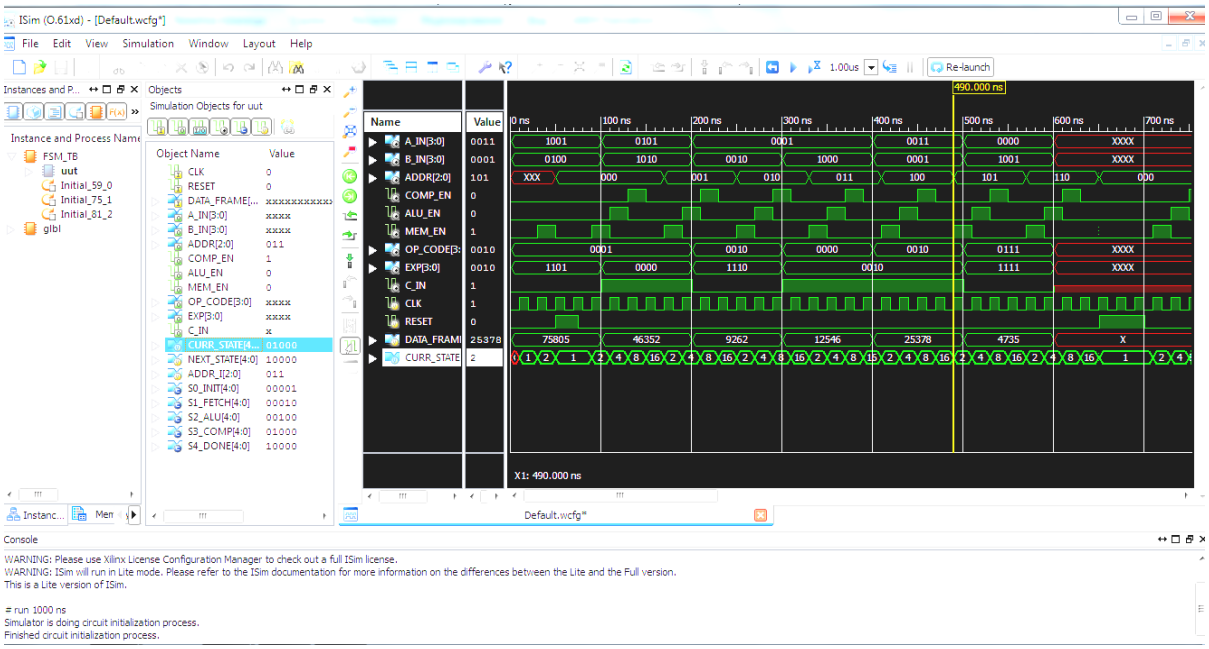


Рисунок 5.5.7 – Назначение внутренних сигналов UUT для отображения во временной диаграмме.

Задание 5.5. Проектирование модуля верхнего уровня «Вычислитель»

Напишите Verilog-код структурного описания модуля ***SIMPLE_CALC***, находящегося на верхнем уровне иерархии проекта «Вычислитель», и состоящего из модулей более низкого уровня, разработанных при выполнении заданий 5.1…5.4. Для включения в объект ***SIMPLE_CALC*** один из объектов ***COMP_BEH*** или ***COMP_RTL*** используйте специальную процедуру **generate,** проверяющую состояние специально введенного в код однобитового флага **`SINTH`**. Измените содержимое памяти **MEM** так, чтобы выполнить проверку работоспособности проекта с помощью испытательного стенда и проверьте работоспособность разработанного описания. Реализуйте схему вычислителя на ПЛИС (FPGA Spartan 6), используя возможности демонстрационной платы ALTYS.

Указания для выполнения задания 5.5:

Структурная схема объекта верхнего уровня SIMPLE_CALC представлена на рисунке 5.5.3. Простейший вычислитель можно представить в видё «чёрного ящика» с входами синхронизации CLK и сброса RESET и одним выходом RESULT, сигнализирующим о правильности работы калькулятора. Работу простейшего вычислителя можно описать следующим образом.

Конечный автомат CNTRL_FSM обращается к модулю памяти MEM по адресу 0, и происходит считывание операндов, кода операции, значения переноса, ожидаемого результата из памяти. Автомат обращается к АЛУ, загружая в него необходимую информацию, в котором выполняется определённая операция. Конечный автомат загру-

жает значение ожидаемого результата в компаратор. Результат из АЛУ подаётся на компаратор, где происходит сравнение ожидаемого результата и значения, выработанного АЛУ. Если они совпали, сигнал RESULT на выходе компаратора принимает значение «1», и «0» - в обратном случае в случае ошибки. Далее автомат переходит в начальное состояние и происходит считывание операндов из памяти по адресу 2 и т.д. Вышеописанные действия выполняются до тех пор, пока не будут обработаны операнды, считанные из памяти по адресу 5. Обращения конечного автомата к модулям MEM, ALU и COMP тактированы синхросигналом CLK. Сигнал RESET служит для сброса конечного автомата в начальное состояние, и считывание из памяти в этом случае начинается вновь по адресу 0. При выполнении этого задания рекомендуется следующая последовательность действий:

1) Создание нового проекта и включение в него файлов MEM.v, COMP.v, ALU.v и FSM.v с кодами устройств, разработанных ранее. Примите во внимание, чтобы обозначения портов ввода и вывода модуля верхнего уровня соответствовали рисунку 5.5.3. Для включения в проект копий составляющих модулей нижнего уровня используйте функции **Project – Add Copy of Source** и опцию **All** (или **Synthesis/Imp + Simulation**) для того, чтобы проводить синтез и моделирование подключенных модулей в данном проекте (см. рисунок .5.5.8).

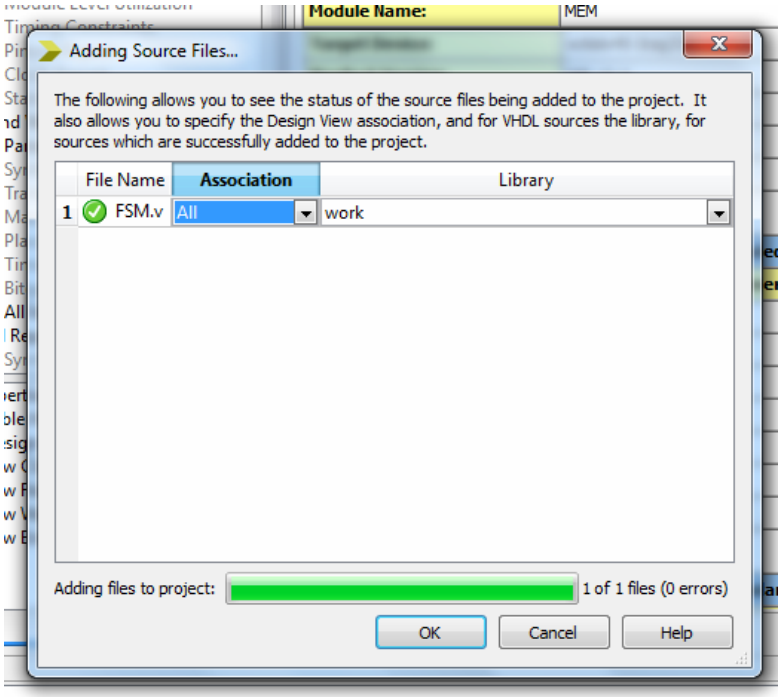


Рисунок 5.5.8 – Окно для дабвления копий уже созданных кодов в проект.

2) Модификация кода модуля MEM, для загрузки тестового содержимого ROM используя данные заданий 5.1 и 5.3.

Для проверки реакции на ошибку внесите изменение в значение одного или нескольких разрядов сигнала EXP, входящего в состав сигнала памяти MEM DATA_FRAME. Например, так:

```
…
3'b011:DATA_FRAME = 17'b0001_0010_0_0010_1010;  // 9258 , а не 9262
…
```

3) Написание Verilog-кода, проверка его синтаксиса и анализ реализации на RTL уровне и в ПЛИС. Пример фрагмента кода, для включения в проект модулей нижнего уровня приведен на рисунке 5.5.9.

```
`timescale 1ns / 1ps
`define SINTH 0
 module SIMPLE_CALC(input CLK, RESET, output RESULT);

wire [3:0] A_IN, B_IN, OP_CODE, EXP_OUT, ALU_OUT ;
wire C_IN, ALU_EN, COMP_EN, MEM_EN ;
wire [2:0] ADDR_SIG ;
wire [16:0] DATA_FRAME ;

 ALU ALU_INST0 (
     .OP_CODE(OP_CODE),
     .A(A_IN),
     .B(B_IN),
     .C_IN(C_IN),
     .EN(ALU_EN),
     .Y(ALU_OUT),
    .CLK(CLK)
    );
 CNTRL_FSM FSM_INST0 (
     .CLK(CLK),
     .RESET(RESET),
     .DATA_FRAME(DATA_FRAME),
     .A_IN(A_IN),
     .B_IN(B_IN),
     .OP_CODE(OP_CODE),
     .EXP(EXP_OUT),
     .ADDR(ADDR_SIG),
     .COMP_EN(COMP_EN),
     .ALU_EN(ALU_EN),
    .MEM_EN(MEM_EN),
     .C_IN(C_IN)
    );
 MEM MEM_INST0 (
     .ADDR(ADDR_SIG),
     .DATA_FRAME(DATA_FRAME),
    .CLK(CLK),
    .EN(MEM_EN)
    );
… // здесь нужно включить процедуру generate
endmodule
```

Рисунок 5.5.9 – Фрагмент кода модуля верхнего уровня проекта «Вычислитель».

Для реализации одного из двух возможных сценариев использования кодов модуля компаратора COMP_BEH или COMP_RTL примените процедуру **generate** проверяющую значение флага ‘**SINTH**:

```
generate

  case (`SINTH )
    1'b0:
    begin : U
COMP_BEH COMP_INST_BEH (CLK, COMP_EN, EXP_OUT, ALU_OUT, RESULT
);
    end
    default:
    begin: K
COMP_RTL COMP_INST_RTL (CLK, COMP_EN, EXP_OUT, ALU_OUT, RESULT
);
    end
  endcase

endgenerate
```

После синтеза следует провести анализ отчета проекта о реализации проекта в конкретном исполнении в ПЛИС, а также просмотреть RTL–реализацию (функция **View RTL Schematic)** и вариант реализации из компонентов ПЛИС (**View Technology Schematic**).

Примеры соответствующих схем приведены на рисунках 5.5.10 и 5.5.11.

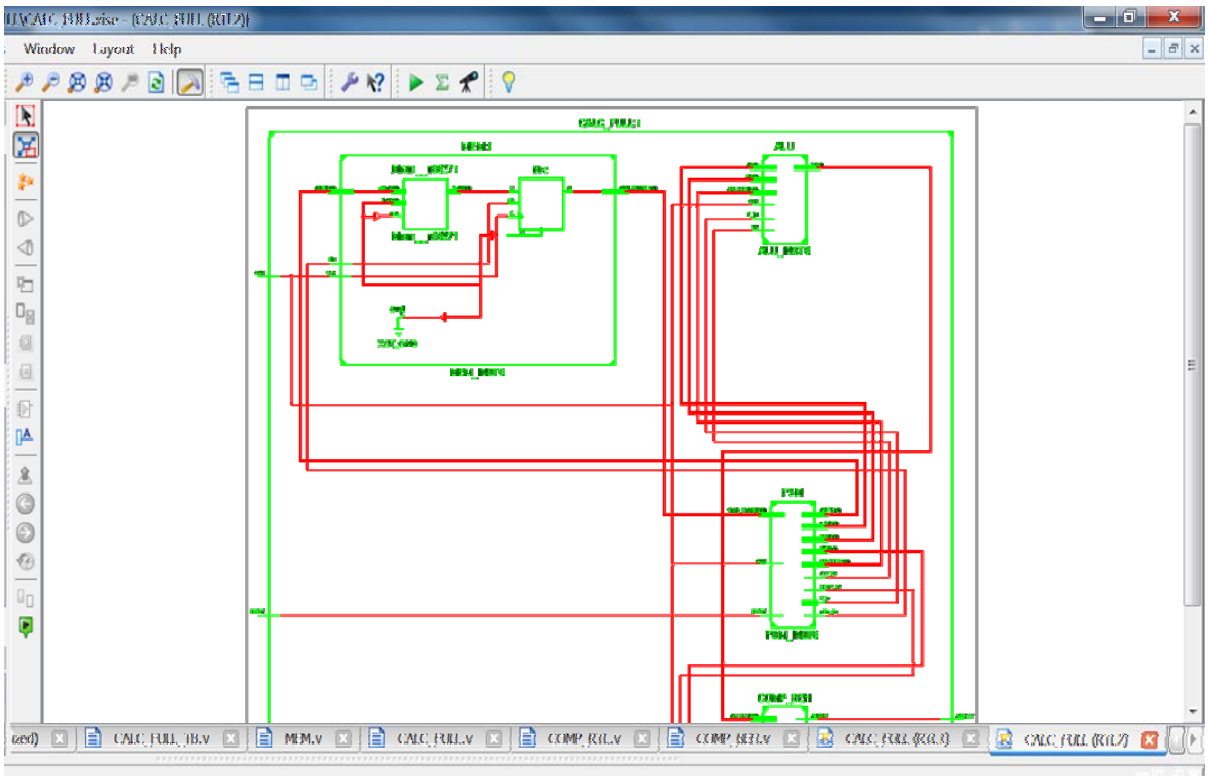


Рисунок 5.5.10 – RTL-схема устройства «Вычислитель».

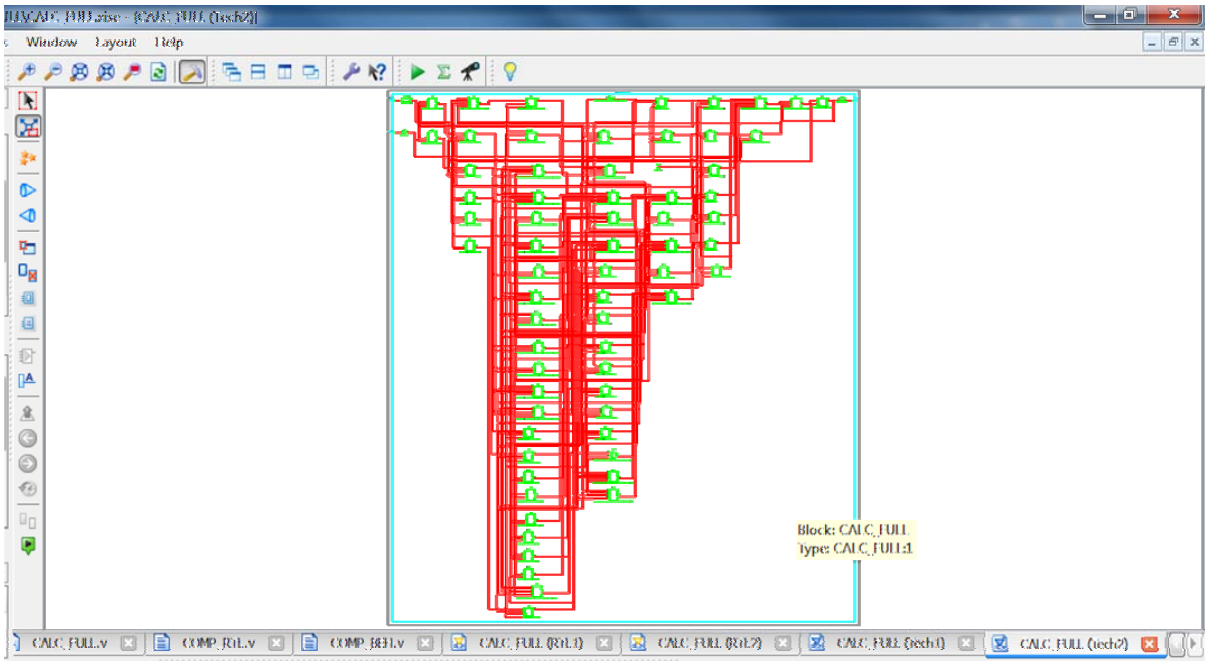


Рисунок 5.5.11 – Схема устройства «Вычислитель», реализованная на ресурсах ПЛИС.

4) Разработка кода испытательного стенда, моделирование и проверка правильности функционирования вычислителя. Пример кода для моделирования работы вычислителя приведен на рисунке 5.5.12.

```
`timescale 1ns / 1ps

module SIMPLE_CALC_TB_v;

   reg CLK_TB = 1'b0;
   reg RESET_TB = 1'b0;

   wire RESULT_TB;

   SIMPLE_CALC uut (
         .CLK(CLK_TB),
         .RESET(RESET_TB),
         .RESULT(RESULT_TB)
   );

   initial
   forever #10 CLK_TB = ~ CLK_TB ;

   initial
   begin
         #10 RESET_TB = 1'b1 ;
         #25 RESET_TB = 1'b0 ;
         #650 RESET_TB = 1'b1 ;
         #25 RESET_TB = 1'b0 ;
   end

endmodule
```

Рисунок 5.5.12 – Пример кода испытательного стенда для вычислителя.

5) Выполнение моделирования и анализ результатов. Пример диаграммы с результатами моделирования работы вычислителя приведен на рисунке 5.5.13.

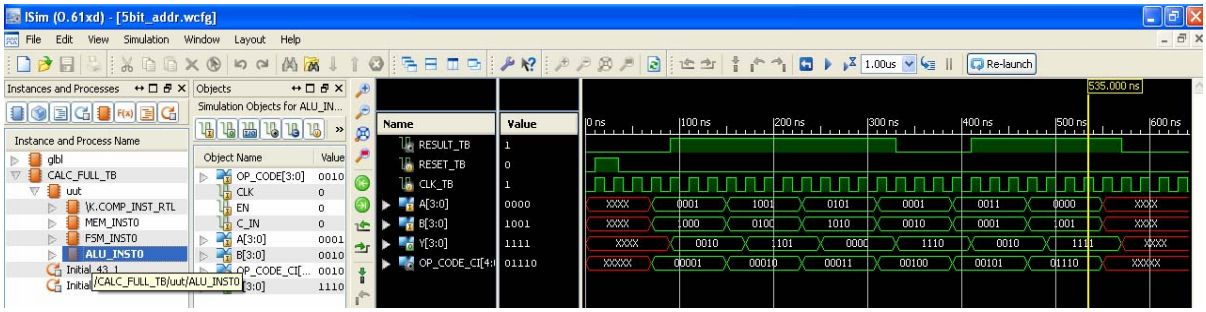


Рисунок 5.5.13 – Окно программы iSim с примером временной диаграммы моделирования работы устройства «Вычислитель».

Кроме отображения входных сигналов CLK и RESET и выходного сигнала RESULT для анализа работы модулей нижнего уровня в диаграмме также назначены для воспроизведения сигналы OP_CODE[3:0], C_IN, OP_CODE_C_I[4:0], A[3:0], B[3:0], Y[3:0]. Для этого нужно открыть модуль UUT на закладке **Instants and Process Name,** перенести нужные сигналы из окна **Objects** в окно **Name**. Из диаграммы моделирования видно:

- считывание данных происходит семь раз, после чего устройство переходит в режим ожидания, и после прихода сигнала RESET, цикл повторяется;
- при считывании седьмого слова данных значения сигналов не определены;
- сигнал RESULT принимает значения логического нуля в период обработки слова памяти, содержащегося по третьему адресу, что соответствует внесенной нами преднамеренно ошибке в значение сигнала EXP;

Таким образом, результаты моделирования позволяют сделать вывод о том, что проект выполнен успешно.

6) На заключительном этапе проекта следует получить данные отчетов о затраченных ресурсах при реализации в ПЛИС. Пример такого отчета приведен в таблице 5.5.6.

7) Для упражнения в качестве дополнительного задания можно реализовать проект в ПЛИС FPGA 6SLX45CS в корпусе G324, используя демонстрационную плату ALTYS. Для этого в файле пользовательских ограничений следует назначить подключение сигнала CLK к выводу15, сигнал RESET – подключить к одной из командных кнопок, а выход RESULT – к одному из светодиодов. После этого следует выполнить процедуру имплементации проекта подобно тому, как это подробно описано в разделе 2 настоящего пособия.

Таблица 5.5.6 – Данные из отчета о ресурсах ПЛИС FPGA 6SLX45CS.

Наименование узла ПЛИС	Использовано	Доступно	% исп.
количество задействованных ячеек Slices Registers	26	54576	~0
количество задействованных блоков Slice LUTs	39	27288	~0
количество полностью задействованных пар с триггерами FF (Slice LUTs-FF)	20	45	44
количество ячеек ввода-вывода (Bonded IOBs)	3	218	~1

ПРИЛОЖЕНИЕ 1. ОПИСАНИЕ ПЛАТЫ ALTYS

Общие сведения

Плата Atlys (рисунок П1.1) - это отладочный набор, основанный на ПЛИС Xilinx Spartan-6 LX45 FPGA.

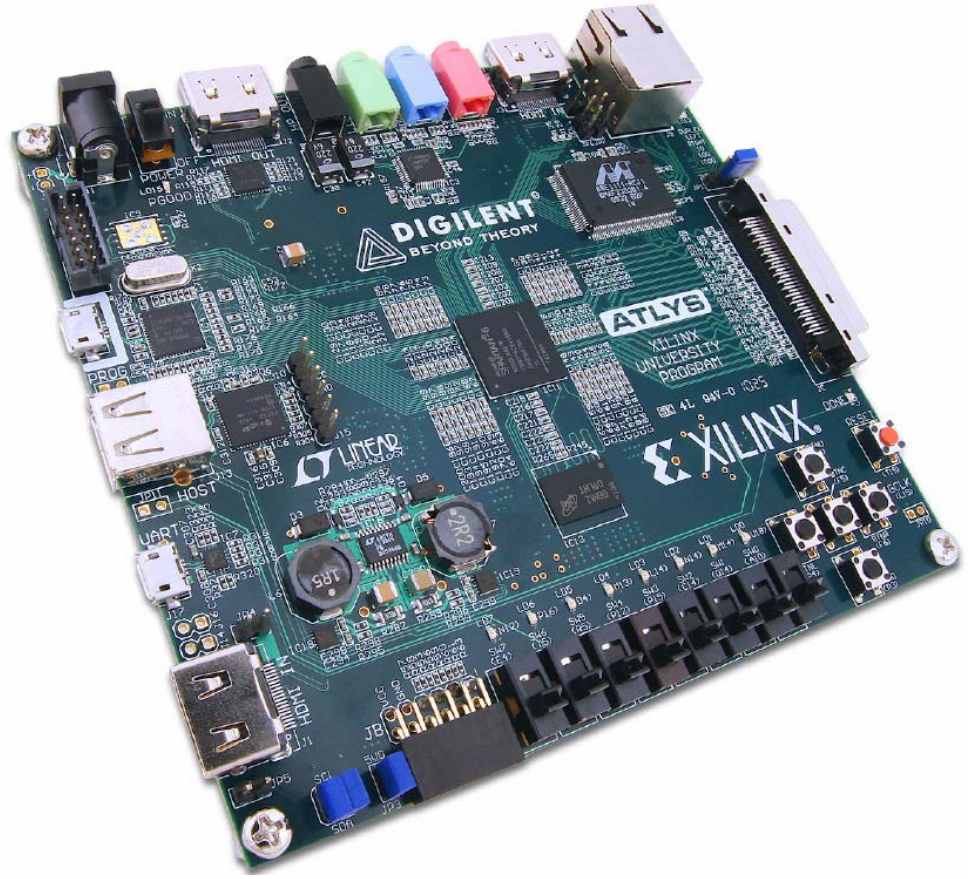


Рисунок П1.1 - Внешний вид платы ALTYS.

Наличие на плате FPGA большого набора периферийных устройств высокого уровня, таких как: Gbit Ethernet, HDMI контроллер, память типа DDR2 (128 Мб, с длинной слова 16 бит), разъемы аудио и USB, делают плату Atlys идеальным управляющим устройством для широкого ряда цифровых систем, включая разработку процессоров основанную на ядре MicroBlaze фирмы Xilinx. Плата Atlys совместима с САПР Xilinx (ChipScope, EDK, WebPack и др.) следовательно, разработка целого ряда устройств на ее основе не требует дополнительных денежных затрат.

Микросхема FPGA Spartan-6 LX45 оптимизирована для реализации устройств высокопроизводительной логики и включает в себя (см. рис.2):

- 6.822 ячейки, каждая из которых содержит четыре 6-ти выводные таблицы преобразования (LUT), и 8 триггеров;
- быстродействующие блоки RAM с общим объемом 2,1 Mbit;
- четыре блока управления синхронизацией (восемь DCM и четыре PLL);

- шесть систем фазовой автоподстройки частоты;
- 58 DSP ячейки;
- тактовая частота порядка 500 МГц.
-

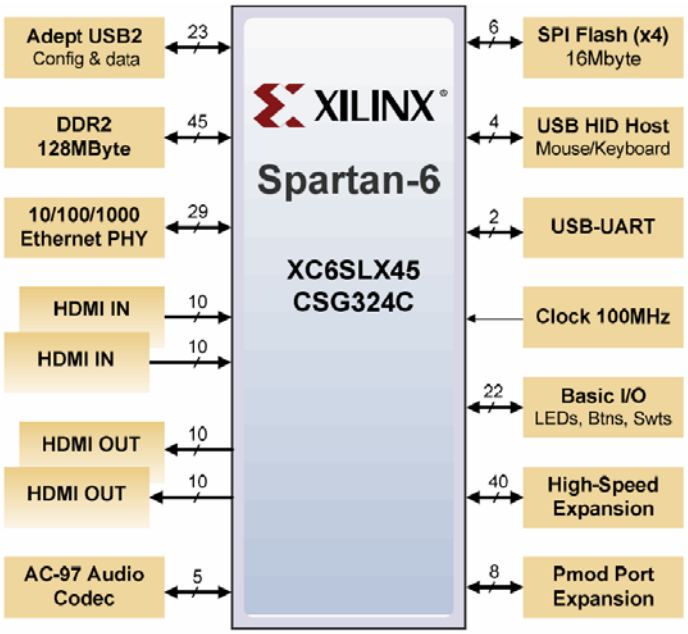


Рисунок П1.2 – Периферийные порты микросхемы FPGA Spartan-6 LX45.

Плата Atlys также включает в себя новую периферийную систему фирмы Digilent, Adept USB2. Даная система предназначена для программирования FPGA, проверки питающих напряжений, автоматическое тестирование платы и осуществления упрощенной передачи пользовательских данных.

Полный набор поддерживаемых платой IP ядер и примеры проектов, а также большой выбор дополнений для платы можно найти на сайте: www.digilentinc.com.

Характеристики платы

- FPGA серии Spartan 6 LX45, фирмы Xilinx, BGA корпус, 324 вывода;
- DDR2 память, объемом 128Мбайт, с длинной слова 16 бит;
- 10/100/1000 Ethernet (протокол физического уровня);
- 2 порта USB для программирования и обмена данными;
- Встроенный USB-UART мост, и USB-HID порт (для “мышки” и клавиатуры);
- Два HDMI входа и выхода;
- AC-97 аудио кодек;
- Контроль питания в режиме реального времени для всех питающих напряжений;
- 16Мбайт x 4 SPI Flash память для конфигурирования и хранении данных;
- 100Мгц КМОП генератор;
- 48 выводов разведены на внешние разъемы;
- GPIO включает в себя 6 кнопок, 8 светодиодов и переключателей.

Порядок настройки платы

После включения платы, FPGA должна быть сконфигурирована, для выполнения каких-либо функций. Конфигурация FPGA может быть выполнена тремя способами (рисунок П1.3):

1. При помощи ПК, подключенного к плате через USB или JTAG разъем;
2. Автоматически при подаче питания при помощи конфигурационного файла, записанного во Flash памяти;
3. При помощи конфигурационного файла, записанного на USB накопитель, подключенный через разъем USB-HID.

Расположенной на плате перемычкой (JP11) осуществляется выбор между JTAG/USB и программированием из ROM. Если перемычка разомкнута, то FPGA автоматически считывает конфигурационный файл, записанный в ROM, при подаче питания на плату. Если перемычка замкнута, то FPGA остается в режиме ожидания пока не будет сконфигурирована по интерфейсу JTAG или последовательному программируемому порту.

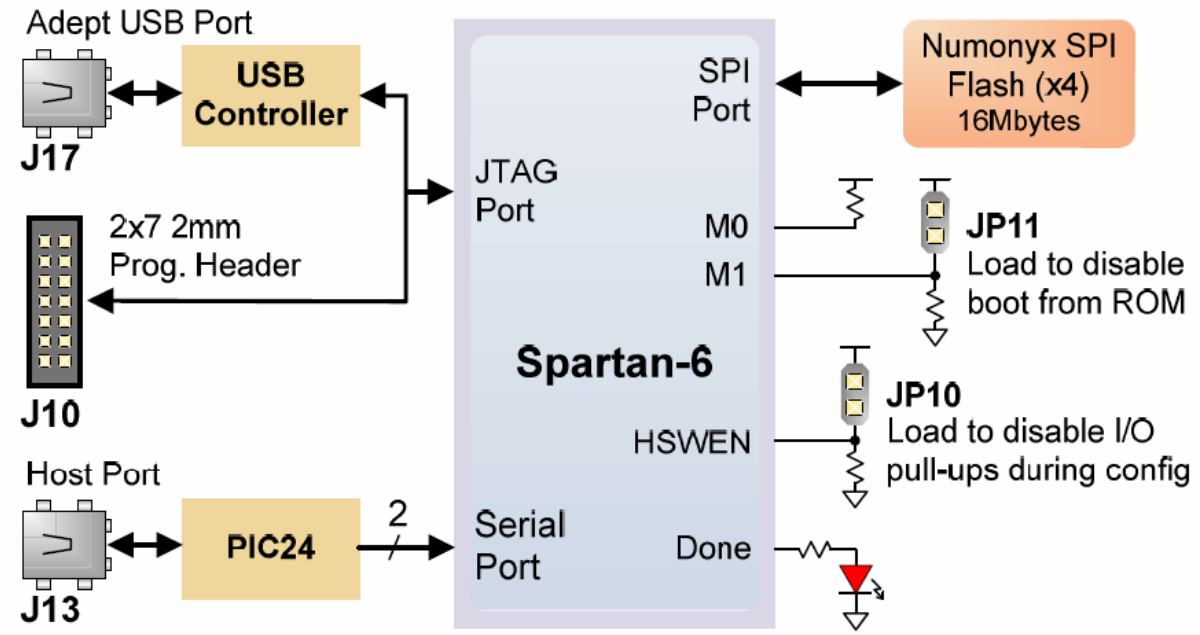


Рисунок П1.3 – Подключение микросхемы FPGA Spartan-6 LX45, расположенной на плате ALTYS к внешним разъемам.

Перемычка JP12 должна быть всегда замкнута (либо на 3.3В, либо на 2.5В). Если перемычка JP12 разомкнута, то банк 2 FPGA недоступен, также не реализуется подтяжка к питанию для сигналов CCLK, DONE, PROGRAM_B и INIT_B. В этом случае FPGA находится в состоянии сброса, и не доступна для конфигурирования по JTAG или из ROM.

Для программирования FPGA и ROM, могут быть использованы бесплатные программы Digilent и Xilinx.

Файл прошивки хранится в FPGA при помощи внутренних ячеек памяти типа SRAM. Данный файл содержит в себе информацию о логических функциях и

соединениях, и остается неизменным, пока не будет отключено питание или не будет перезаписан другим файлом.

Через интерфейс JTAG передаются конфигурационные файлы типа *.bin или *.svf. Через USB интерфейс – только *.bit файлы, в ROM память могут быть записаны файлы *.bin, *.bit, *.svf и *.mcs.

Файлы конфигурации могут быть созданы как при помощи WebPack, так и EDK из исходных файлов, описанных при помощи VHDL, Verilog или схематического представления. EDK может быть использовано для проектирования процессорных систем. Программирования FPGA и ROM выполняются при помощи программы iMPACT или Adept, в последнем случае - через порт, обозначенный как Adept USB (маркировка на плате “Prog”).

При программировании FPGA файлы *.bit или *.svf записываются непосредственно в память FPGA через JTAG-USB порт. Программирование ROM памяти осуществляется в два этапа. Сначала программируется FPGA схемой программирования ROM памяти по интерфейсу SPI, а затем данные передаются в ROM через FPGA (данный процесс выполняется при выборе в меню “program ROM”). После записи конфигурационного файла в ROM, FPGA будет загружена автоматически при следующем включении или после сброса, если перемычка JP11 не установлена (см. рисунок П1.4).

FPGA может быть сконфигурирована при помощи карты памяти, подключенной к разъему USB-HID. Для такого конфигурирования необходимо выполнение условий, чтобы в ее корневой директории был записан только один *.bit файл, перемычка JP11 установлена и подключено питание.

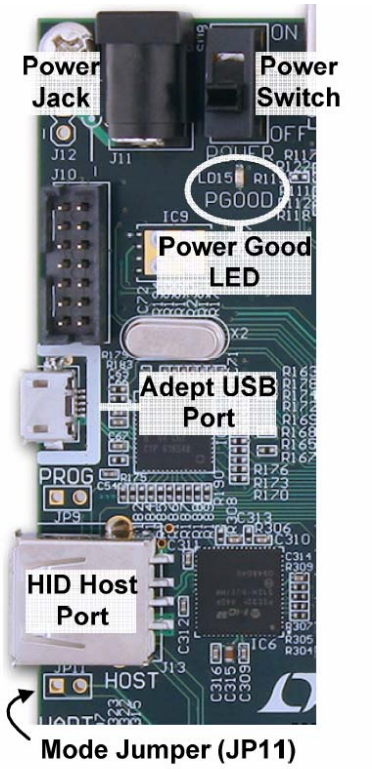


Рисунок П1.4 – Расположение некоторых органов управления и перемычек на плате.

Порт Adept USB для инструментов iMPACT

Порт Adept USB (см. рисунок П1.4) совместим с программой iMPACT фирмы Xilinx, если на управляющем компьютере установлено дополнение от Digilent. Данное дополнение автоматически переводит iMPACT-команды JTAG в формат, совместимый с USB портом от Digilent.

Для программировании FPGA через программу Adept главное окно которой изображено на рисунке П1.5, необходимо выбрать вкладку Config, и убедиться в том, что плата Atlys подключена и FPGA найдена. При помощи кнопки Brows необходимо выбрать файлы прошивки, а затем нажать кнопку Program.

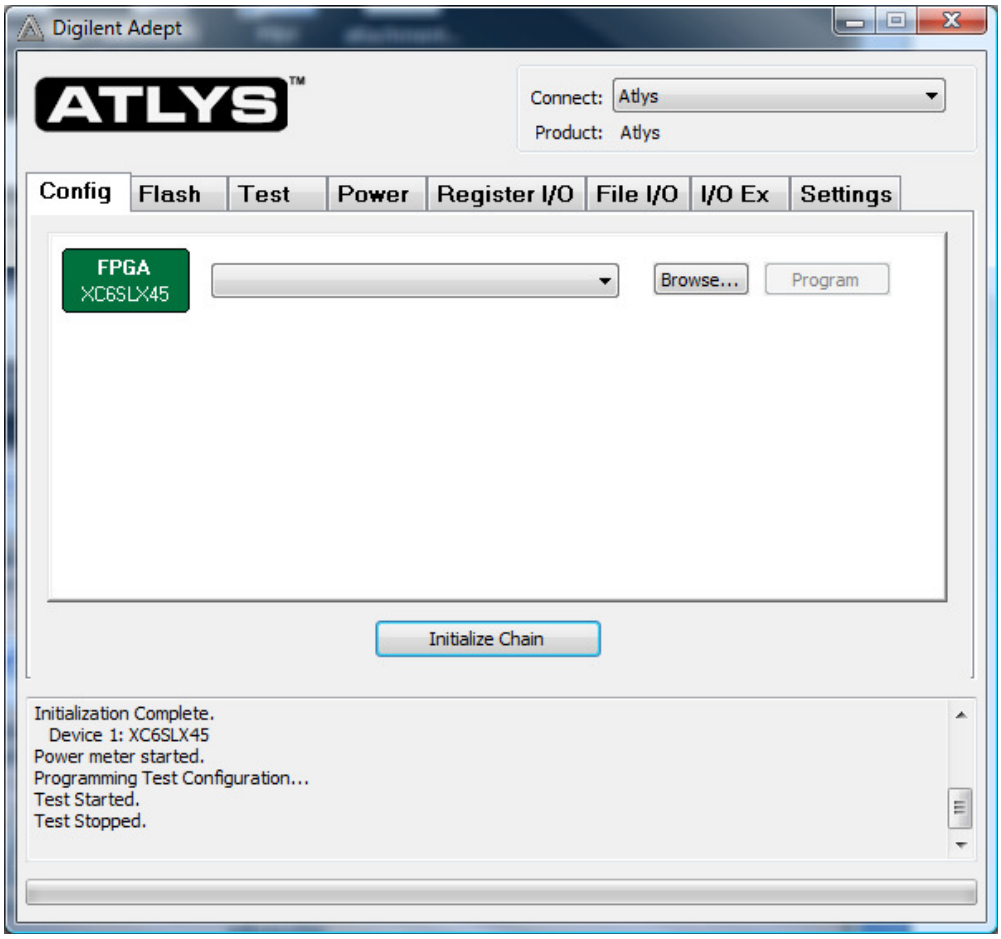


Рисунок П1.5 – Основное окно программы ADEPT,
предназначенной для работы с платой ALTYS

Источник питания

Плата Atlys требует внешнего подключения источника 5В питания, рассчитанного на нагрузку до 4А (адаптер для сети 220 В входит в комплект поставки). На плате используются преобразователи напряжения от фирмы Linear Tec. для формирования требуемых напряжений (см. таблицу П1.1 и рисунок П1.6).

Таблица П1.1 – Преобразователи напряжения, установленные на плате и их нагрузочные характеристики.

Напряжение	Схема	Устройство	Ток (макс./тип.)
3,3 В	FPGA I/O, видео, USB порт, тактирование, ROM, аудио	IC16: LT3501	3A / 900mA
2,5 В	FPGA aux, VHDC, Ethernet PHY I/O, GPIO	IC15: LTC3546	1A / 400mA
1,2 В	ядра FPGA и Ethernet PHY	IC15: LTC3546	3A / 0.8 – 1.8A
1,8 В	DDR и выводы FPGA DDR	IC16: LT3501	3A / 0.5 -- 1.2A
0,9 В	Ограничивающее напряжение DDR	IC14: LTC3413	3A / 900mA

Контроль тока четырех первых напряжений на плате Atlys осуществляется при помощи 16-ти битного дельта-сигма АЦП LTC2481. С точностью в 1% измеренные величины можно посмотреть в программе Adept.

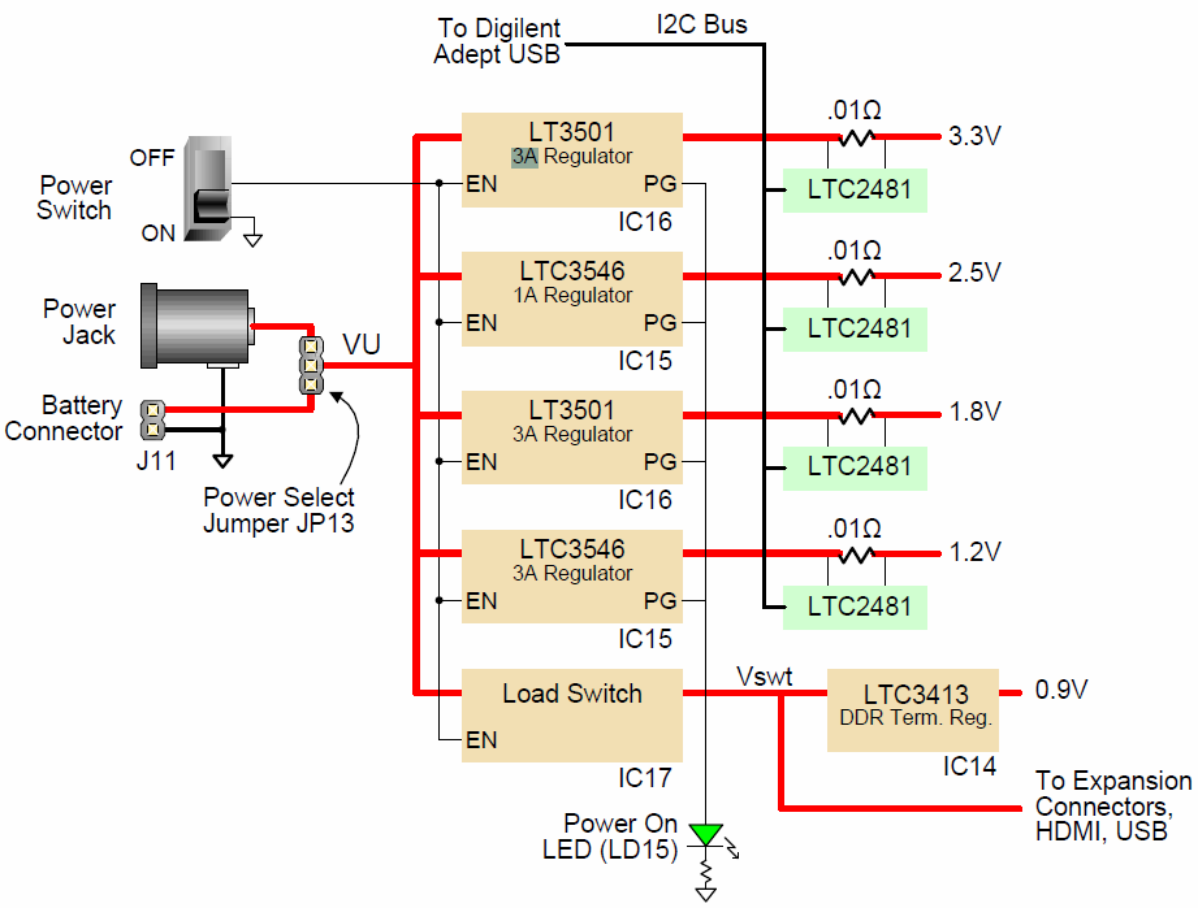


Рисунок П1.6 – Преобразователи напряжения, установленные на плате.

Управление подачей напряжения осуществляется при помощи переключателя SW8. Светодиод LD15 подключен по схеме монтажное «ИЛИ» ко всем тестовым выходам микросхем преобразователей напряжения, для индикации того, что погрешность выходного напряжения каждого источника не превышает 10% от номинального значения.

Микросхема IC17 (FDC6330) пропускает входное напряжение VU на узел Vswt, когда переключатель SW8 находиться в разрешающем положении.

Напряжение Vswt используется различными системами на плате, такими как порт HDMI, шина I2C и USB host.

Шина Vswt также выведена в разъеме расширяющем возможности платы, поэтому напряжение на дополнительных платах может подаваться одновременно с включением платы Atlys.

Память DDR2

На плате установлена DDR2 память объемом 1Гбит, управляемая при помощи блока контроля DDR в FPGA. DDR2-память, эквивалентная MT47H64M16-25E, содержит 16-ти битную шину и 64M ячеек. Обмен данными с DDR2 гарантируется на частотах до 800 МГц. Интерфейс подключения DDR2-памяти, и разводка монтажных дорожек осуществлена согласно руководству пользователя «Xilinx Memory Interface Generator (MIG)» (см. рисунок П1.7).

Интерфейс памяти поддерживает SSTL18 сигналы, и все адреса, данные, тактовые импульсы и управляющие сигналы разведены с учетом задержек и импеданса.

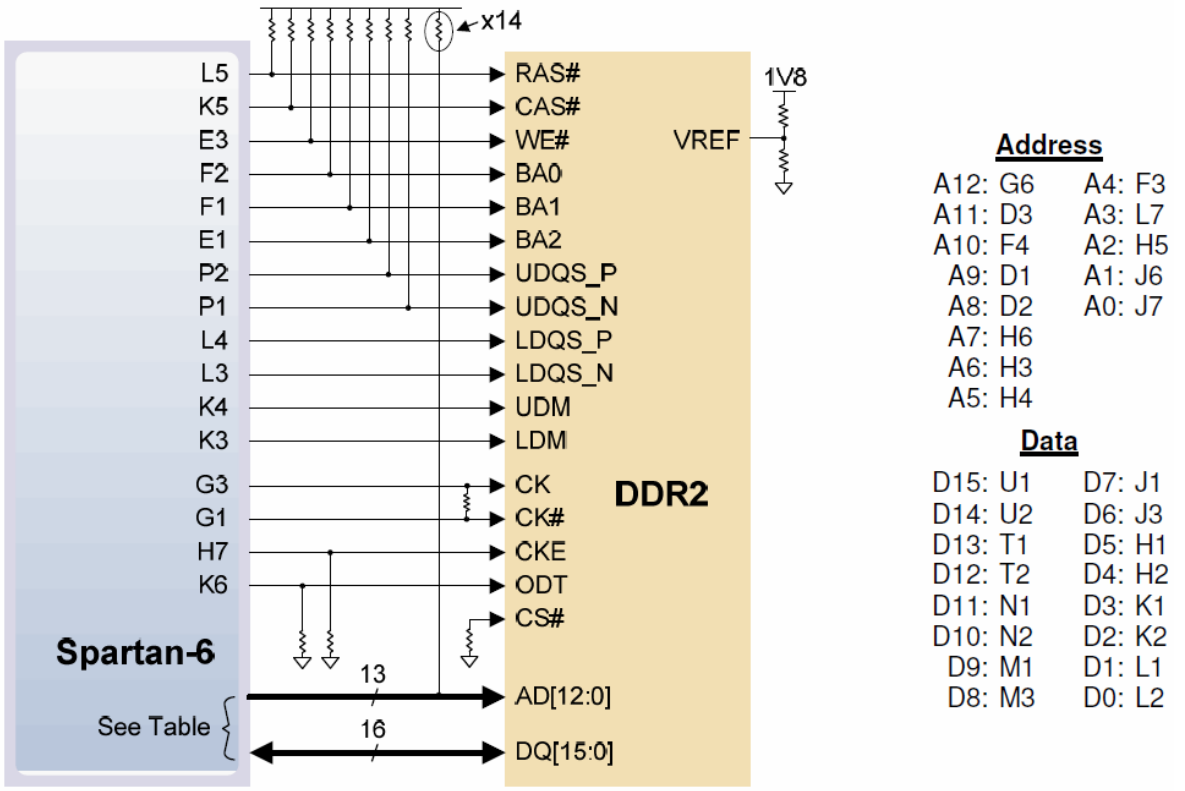


Рисунок П1.7 – Подключение DDR2 памяти к Spartan-6 на плате ALTYS.

Адресная шина и управляющие сигналы подключены через ограничительные сопротивления 47 Ом к ограничивающему напряжению 0.9В, а шина данных подключается при помощи On-Die-Termination (ограничивающие сопротивления размещенные внутри микросхемы). Для синхронизации DDR-памяти в FPGA, специально предусмотрена пара согласованных тактовых сигналов в DDR2, передние фронты которых сделаны пологими.

Флеш память

На плате Atlys установлена флеш-память Numonyx N25Q12 (128 Мбит), предназначенная для хранения конфигурационного файла и данных (см. схему подключения на рисунке П1.8. Файлы конфигурации занимают 12 Мбит памяти, остальные 116 Мбит могут использоваться для хранения данных. Обмен данными между ПК и флеш-памятью может быть осуществлен при помощи пользовательской программы, или при помощи программы, сгенерированной программой Adept. Пользовательская программа позволяет настроить обмен данными между FPGA и ROM. На сайте Digilent Вы можете найти проект, реализующий такой обмен данными.

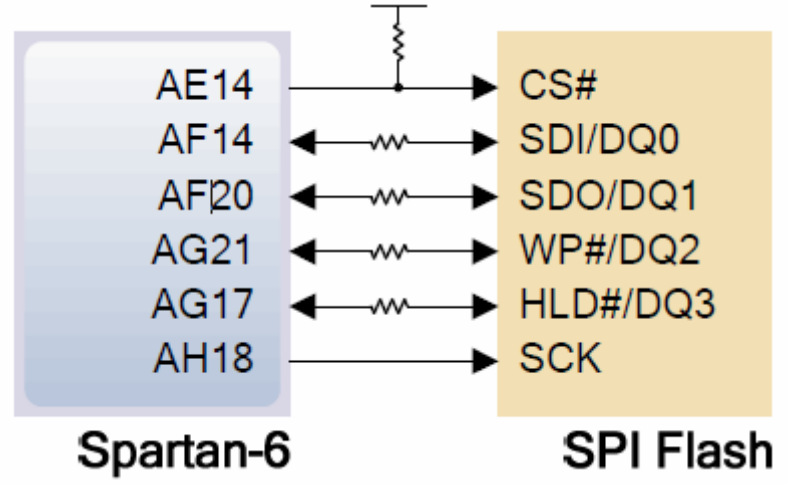


Рисунок П1.8 - Рисунок П1.7 – Подключение флеш памяти к Spartan-6 на плате ALTYS.

Сеть Ethernet PHY

Плата Atlys содержит контроллер Marvell Alaska Tri-mode PHY (the 88E1111) соединенный с разъемом Halo HFJ11-1G01E RJ-45 (рисунок П1.9). Оба устройства MII и GMII поддерживают режимы обмена данными на скоростях 10/100/1000 Мбит/с.

Настройки по умолчанию, используемые при включении питания или сбросе:

• Режим MII/GMII для обмена данными по медному кабелю.

• Включено автоматическое согласование режима работы с поддержкой всех скоростей, предпочтительно ведомое устройство.

• Выбран интерфейс MDIO, адрес PHY MDIO = 00111.

• Нет асимметричной паузы, нет MAC паузы, автоматическое определение перекрестного соединения.

• Функция Energy detect отключена (Отключен «Режим сна»).

Для получения более подробной информации по Marvell PHY необходимо связаться с производителем и заключить соглашение о неразглашении.

Проекты, основанные на Xilinx Embedded Developer Kit (EDK), могут получить доступ к PHY, используя IP ядра **xps_ethernetlite** для проектов со скоростями 10/100 Мбит/с и xps_ll_temac для проектов со скоростями 10/100/1000 Мбит/с. IP-ядро xps_ll_temac использует аппаратное ядро MAC Ethernet, включенное в FPGA Virtex-5.

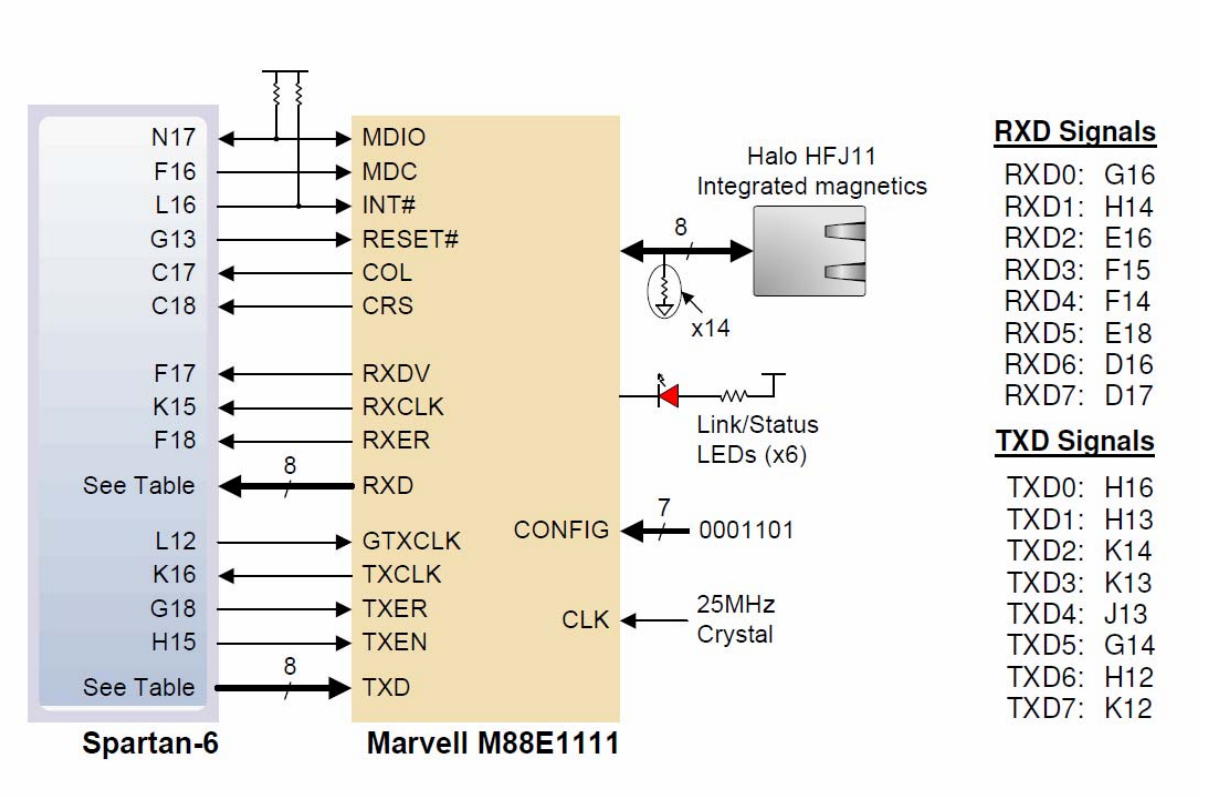


Рисунок П1.9 – Подключение к контроллеру Marvell Alaska Tri-mode PHY (the 88E1111).

Пакет поддержки Atlys Base System Builder (BSB) автоматически генерирует тестовые приложения для MAC Ethernet, которые могут использоваться в качестве справочной информации при создании собственных проектов. Другой демонстрационный проект (веб-сервер), основанный на Ethernet, может быть получен на веб-сайте Digilent.

В проектах ISE может использоваться генератор IP ядер для создания IP ядра контроллера tri-mode Ethernet MAC.

Подключение к видеосистемам (HDMI Ports)

На рисунке п1.10 показано, что плата Atlys содержит четыре порта HDMI, включая два буферизированных порта ввода/вывода HDMI, один буферизированный выходной порт HDMI, и один небуферизованный порт, который может быть как вводом, так и выводом (обычно используемый в качестве выходного порта). Три буферизированных HDMI порта используют разъемы типа A, а небуферизированный

порт использует разъем типа D, расположенный на нижней стороне печатной платы, сразу под разъемом Pmod (разъем типа D намного меньше, чем разъем типа A). Шина данных небуферизированного порта используется совместно с разъем Pmod. Это немного ограничивает пропускную способность, поскольку несколько совместно используемых разъемов не смогут передавать или получать высокочастотные видеосигналы по длинным кабелям HDMI.

Так как HDMI и DVI используют одинаковый TMDS стандарт передачи данных, для управления разъемом DVI через любой из выходных портов HDMI может использоваться обычный переходник (доступный в большинстве магазинов электроники). Порт HDMI не поддерживает сигналы VGA, таким образом, аналоговые дисплеи использоваться не могут.

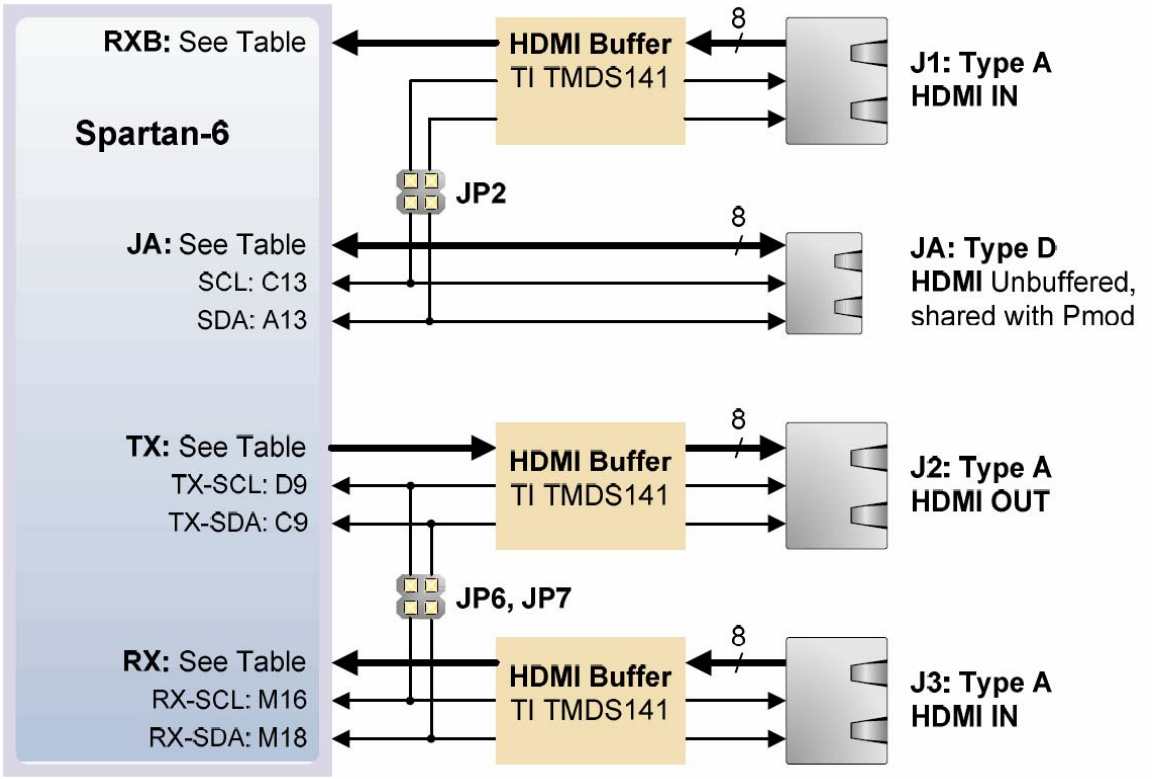


Рисунок П1.10 – Подключение к видеосистемам.

19-контактные разъемы HDMI содержат четыре дифференциальных канала данных, пять контактов GND, один провод протокола дистанционного управления Consumer Electronics Control (CEC), двухпроводный интерфейс Display Data Channel (DDC), который является по существу шиной I2C, сигнал Hot Plug Detect (HPD), 5V 50mA провод питания, и один резервный (RES) контакт. Из них только дифференциальные каналы данных и шина I2C соединяются с FPGA. Описания всех сигнальных контактов приведены на рисунке п1.11.

Проекты, основанные на Xilinx Embedded Developer Kit (EDK) могут использовать xps_tft IP ядро (и его драйвер) чтобы получить доступ к портам HDMI. Ядро xps_tft читает видеоданные из памяти DDR2, и отправляет их по порту HDMI для отображения на внешнем мониторе.

HDMI Type A Connectors				HDMI Type D	
Pin/Signal	J1: IN	J2: Out	J3: IN	Pin/Signal	JA: BiDi
1:D2+	B12	B8	J16	1:HPD	*JP3**
2:D2_S	*GND*	*GND*	*GND*	2:RES	*VCCB2*
3:D2-	A12	A8	J18	3:D2+	N5
4:D1+	B11	C7	L17	4:D2_S	*GND*
5:D1_S	*GND*	*GND*	*GND*	5:D2-	P6
6:D1-	A11	A7	L18	6:D1+	T4
7:D0+	G9	D8	K17	7:D1_S	*GND*
8:D0_S	*GND*	*GND*	*GND*	8:D1-	V4
9:D0-	F9	C8	K18	9:D0+	R3
10:Clk+	D11	B6	H17	10:D0_S	*GND*
11:Clk_S	*GND*	*GND*	*GND*	11:D0-	T3
12:Clk-	C11	A6	H18	12:Clk+	T9
13:CEC	*NC*	*0K to Gnd*	*NC*	13:Clk_S	*GND*
14:RES	*NC*	*NC*	*NC*	14:Clk-	V9
15:SCL	C13	D9	M16	15:CEC	*VCCB2*
16:SDA	A13	C9	M18	16:Gnd	*GND*
17:Gnd	*GND*	*GND*	*GND*	17:SCL	C13**
18:5V	*JP4**	*5V*	*JP8**	18:SCA	A13**
19:HPD	*1K to 5V*	*NC*	*1K to 5V*	19:5V	*JP3*

*jumper can disconnect Vdd **shared with J1 I2C signals via jumper JP2

Рисунок П1.11 – Описания контактов разъемов подключения к видеосистемам.

Учебный проект EDK, доступный на веб-сайте Digilent, выводит на экран градиентную цветовую полосу на мониторе, соединенном с HDMI. Второй учебный проект EDK также, выводит на экран градиентную цветовую полосу на мониторе, соединенном с J2 HDMI.

Аудиокодек (AC-97)

Плата Atlys содержит аудио кодек National Semiconductor LM4550 AC ‘97 (IC19) с четырьмя 1/8” аудио разъемами для: line-out (J16), headphoneout (J18), line-in (J15), и microphone-in (J17) (рисунок п1.12). Поддерживается аудио поток с выборкой до 18 бит на частоте 48 кГц. Частоты дискретизации записи и воспроизведения могут отличаться. Разъем микрофона поддерживает только режим моно, все другие разъемы - стерео. Гнездо наушников соединено с внутренним усилителем аудиокодека на 50 мВт.

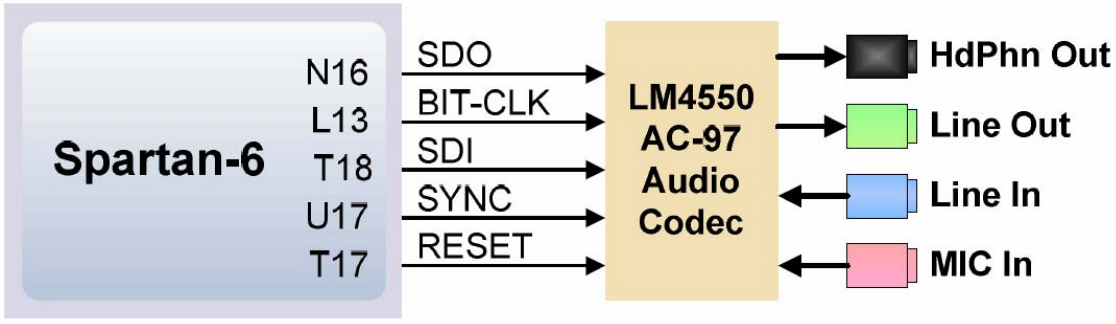


Рисунок П1.12 – Подключение аудиокодека.

Аудиокодек LM4550 совместим со стандартом AC ‘97 v2.1 (Intel) и является основным кодеком (ID1 = 0, ID0 = 0). В таблице П1.2 перечислены характеристики управляющих сигналов кодека AC‘97 и сигналов данных. Все уровни сигналов - LVCMOS33.

Таблица П1.2 –Сигналы и данные аудиосистемы ALTYS

Signal Name	**FPGA Pin**	**Pin Function**
AUD-BIT-CLK	AH17	Последовательный выход тактового сигнала на частоте 12.288 МГц, является половиной частоты 24.576 МГц тактового генератора на входе (XTL_IN).
AUD-SDI	AE18	Последовательный порт данных Serial Data In (к FPGA) от кодека. Данные SDI представляют собой входные кадры данных AC ’97, которые содержат служебные и аудио данные PCM. Данные SDI выдаются по нарастающему фронту AUD-BIT-CLK.
AUD-SDO	AG20	Последовательный порт данных Serial Data Out (к кодеку) от FPGA. Данные SDO представляют собой выходные кадры данных AC ’97, которые содержат служебные и аудио данные DAC. Дискретизация данных SDO происходит по нарастающему фронту AUD-BIT-CLK при помощи LM4550.
AUD-SYNC	J9	Маркер кадра данных AC Link и программный сброс. SYNC (вход кодека) задает границы кадра данных AC Link. Каждый кадр длится 256 периодов AUD-BIT-CLK. Обычно SYNC это положительные импульсы с частотой 48 кГц и скважностью 6.25% (16/256). Сигнал SYNC выдается по нарастающему фронту AUDBIT- CLK, и кодек воспринимает первый импульс SYNC как начало нового кадра данных AC Link. Если последующий синхроимпульс появится в течение 255 периодов AUD-BIT-CLK после начала кадра данных,то он будет проигнорирован. SYNC (активен в состоянии лог. «1») также используется в качестве входа для выполнения асинхронного программного сброса. Программный сброс используется для выхода из состояния выключения питания в интерфейсе AC Link.
AUD-RESET	E12	Аппаратный сброс. Этот сигнал (активен в состоянии лог. «0») вызывает аппаратный сброс, который возвращает регистры команд и все внутренние схемы в их состояние по умолчанию. AUD-RESET должен быть использован для инициализации LM4550 после включения питания, когда напряжение стабилизировались. Также AUD-RESET очищает кодек от тестовых режимов ATE и поставщика. Кроме того, когда AUD-RESET активен, моно вход PC_BEEP подключается непосредственно на оба канала стерео выхода LINE_OUT.

Демонстрационный проект EDK, доступный на веб-сайте Digilent, выводит прямоугольный сигнал в левый канал, правый канал, и затем - на оба звуковых канала разъема LINE OUT. Так же происходит соединение входа LINE IN на выход наушников, с периодической сменой правого и левого каналов.

Демонстрационный проект ISE просто включает звук каналов кодека и подсоединяет LINE IN на разъем LINE OUT.

Тактовый генератор и сигналы синхронизации Oscillators/Clocks

Плата Atlys содержит один CMOS 100 МГц тактовый генератор, соединенный с контактом L15 (L15 - вход GCLK в группе 1). Этот вход тактового генератора может быть подключен к любой из четырех (или ко всем сразу) ячейкам управления синхронизацией в Spartan-6. Каждая ячейка содержит два цифровых блока управления синхронизацией Digital Clock Manager (DCM) и четыре цепи фазовой автоподстройки частоты Phase-Locked Loop (PLL).

DCM поддерживает четыре фазы входной частоты (0., 90., 180., и 270.), деление частоты синхросигнала, (делителем входной частоты может быть любое целое число от 2 до 16 или 1.5, 2.5, 3.5... 7.5) и два противофазных выхода синхросигнала, частота которых может быть умножена на любое целое число от 2 до 32, и также одновременно разделена на любое целое число от 1 до 32.

PLL использует генератор, управляемый напряжением Voltage Controlled Oscillator (VCO), который может быть запрограммирован для формирования сигнала с частотой в диапазоне от 400 до 1080 МГц, путем установки трех наборов программируемых делителей во время конфигурации FPAG. У выхода VCO есть восемь равномерно распределенных выводов (0°, 45°, 90°, 135°, 180°, 225°, 270°, и 315°), частота сигнала с которых может быть разделена на любое целое число между 1 и 128.

Подключение последовательного порта (мост USB-UART)

Плата Atlys содержит USB-UART мост EXAR, чтобы обеспечить возможность приложениям PC связываться с платой, используя COM-порт. Свободно-распространяемые драйвера позволяют легко передавать данные от COM-порта PC на плату Atlys, с использованием USB порта J17 с пометкой «UART» (см. рисунок П1.13). EXAR передает данные в Spartan-6, используя двухпроводной последовательный порт с программным управлением потоком данных (XON/XOFF).

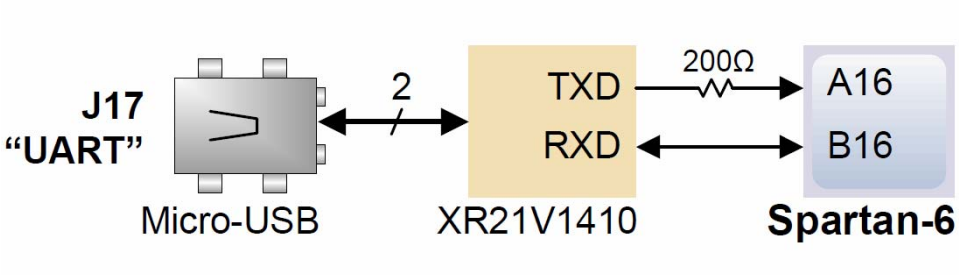


Рисунок П1.13 – Подключение последовательного порта.

Распространяемые свободно драйверы под Windows и Linux могут быть загружены с сайта www.exar.com. Ввод номера компонента "XR21V1410" в поле поиска позволит получить ссылку к странице продукта XR21V1410, где можно найти ссылки для скачивания последних версий драйверов. После того, как драйвера будут установлены, команды ввода-вывода от PC, направленные к COM-порту, так же будут направлены на выводы A16 и B16 FPGA.

Реализация режима контроллер USB HID Host

Установленный на плате микроконтроллер Microchip PIC24FJ192 дает плате Atlys возможность быть HID USB контроллером (рисунок П1.14). Прошивка микроконтроллера MCU позволяет использовать мышь или клавиатуру, подключенную к USB разъему типа A (J13) с маркировкой "Host". В настоящий момент отсутствует поддержка USB концентраторов, и только одно устройство (мышь или клавиатура) может быть подключено одновременно. PIC24 использует четыре сигнала для связи с FPGA: два используются как порт клавиатуры со связью по протоколу PS/2 (специализированному для клавиатуры, см. рисунок П1.15) и два используется в качестве порта мыши со связью по протоколу PS/2 (специализированному для работы с манипуляторами «мышь»).

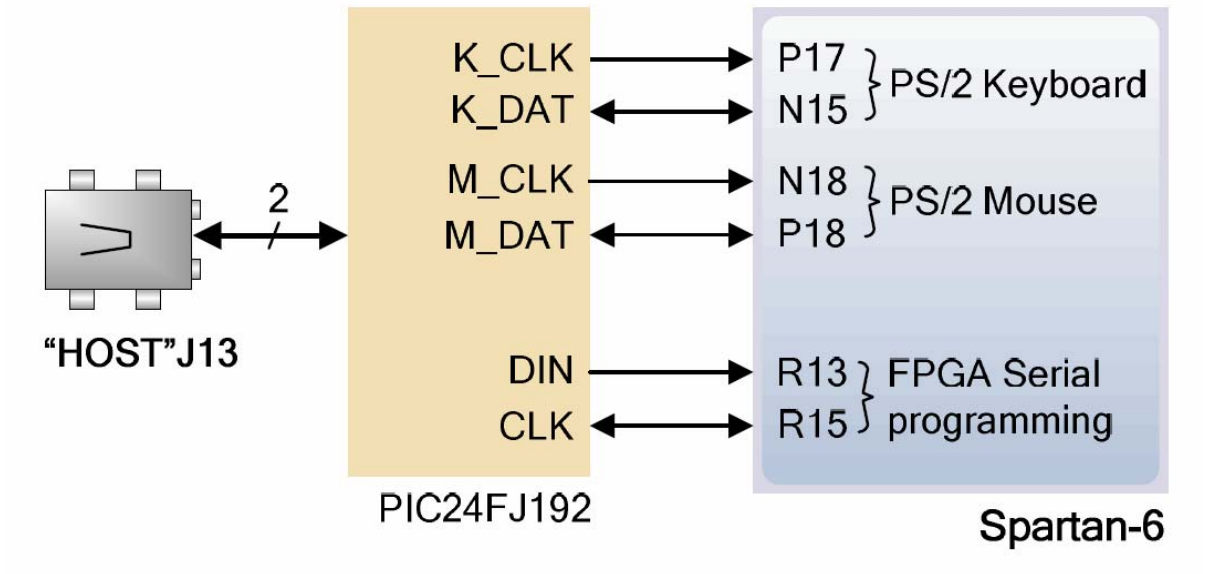
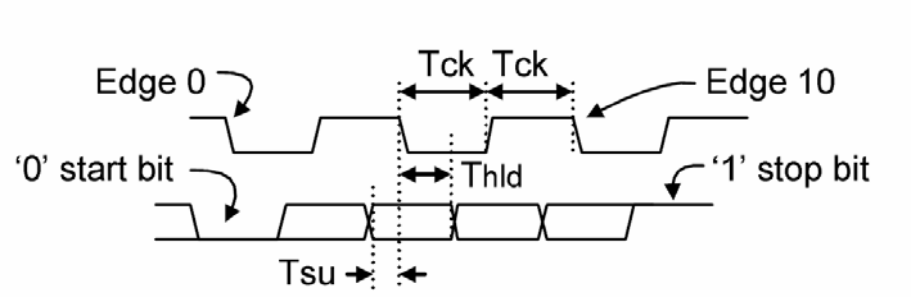


Рисунок П1.14 – Реализация режима «HID USB контроллер».

Два порта ввода-вывода PIC24 также соединяются с последовательным двухпроводным портом программирования FPGA, таким образом, микросхема FPGA может быть запрограммирована данными из файла, хранящегося на USB карте памяти. Чтобы запрограммировать FPGA, подключите карту памяти, содержащую единственный файл программы (***.bit**) в корневом каталоге, замкните JP11, и включите питание платы. Это запустит процесс программирования FPGA при помощи PIC24. В процессе программирования любые некорректные битовые файлы будут автоматически пропущены.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Рисунок П1.15 –Сигналы для обмена по протоколу PS/2.

Чтобы получить доступ к хост-контроллеру USB, проекты EDK могут использовать стандартное PS/2 IP ядро. В демонстрационных проектах, доступных на веб-сайте Digilent, приведен пример считывания символов с USB клавиатуры, подключенной к USB хост-контроллеру.

Мыши и клавиатуры, которые используют протокол PS/2, используют двухпроводную последовательную шину (синхронизация и данные), чтобы связаться с контроллером. И те и другие используют 11-разрядные слова, которые включают стартовый бит, стоповый бит и бит четности, но пакеты данных имеют различную структуру. Интерфейс клавиатуры поддерживает двунаправленную передачу данных (таким образом, контроллер может зажигать светодиоды состояния на клавиатуре). Временные диаграммы работы шины показаны на рисунке. Синхросигналы и сигналы данных изменяются только тогда, когда происходит передача данных. Все остальное время они находятся в состоянии ожидания (лог. '1'). Так же на временных диаграммах приведены требования к сигналам данных и синхронизации. Схема интерфейса PS/2 может быть реализована в FPGA, чтобы создать интерфейсы мыши или клавиатуры.

Подключение клавиатуры

Клавиатура использует схему с открытым коллектором. Таким образом клавиатура или контроллер USB могут управлять двухпроводной шиной (если устройство узла не будет отправлять данные клавиатуре, то узел может использовать порты только для ввода).

PS/2 клавиатуры используют коды сканирования для передачи данных о нажатии клавиши. Каждой клавише присвоен код, который отправляется всякий раз, когда клавиша нажимается. Если клавиша будет удерживаться, то код сканирования будет передаваться повторно, примерно раз в 100 мс. Когда клавиша будет отпущена, одновременно с кодом сканирования данной клавиши будет передан код F0. Если регистр данной клавиши может быть переключен для получения другого символа (такого как прописная буква), то символ смены регистра отправляется в дополнение к коду сканирования, и ведущее устройство должно определить, какой ASCII символ

использовать. Некоторые клавиши, называемые клавишами расширения, отправляют код E0 перед кодом сканирования (и могут отправить более одного кода сканирования). Когда клавиша расширения отпускается, отправляется код клавиши E0 F0, сопровождаемый кодом сканирования. Коды сканирования для большинства клавиш показаны на рисунке. Ведущее устройство так же может отправлять данные клавиатуре. Краткий список команд, которые могут быть отправлены ведущим устройством, представлен в таблице П1.3.

Таблица П1.3 – Команды обмена данными с клавиатурой.

ED	Установка Num Lock, Caps Lock, и Scroll Lock светодиодов. После приема ED, клавиатура возвращает FA, а затем ведущее устройство отправляет байт установки статуса светодиода: бит 0 устанавливает Scroll Lock, бит 1 устанавливает Num Lock, и бит 2 устанавливает Caps lock. Биты с 3 по 7 игнорируются.
EE	Эхо (тест). Клавиатура возвращает EE после приема EE.
F3	Установка частоты повторения кода сканирования. После приема FA, клавиатура возвращает F3, затем ведущее устройство посылает второй байт для установки частоты повторения сканирования.
FE	Повторная передача. FE заставляет клавиатуру повторно передать последний отправленный код сканирования.
FF	Сброс. Сброс клавиатуры.

Коды сканирования для большинства PS/2 клавиатур приведены на рисунке П1.16.

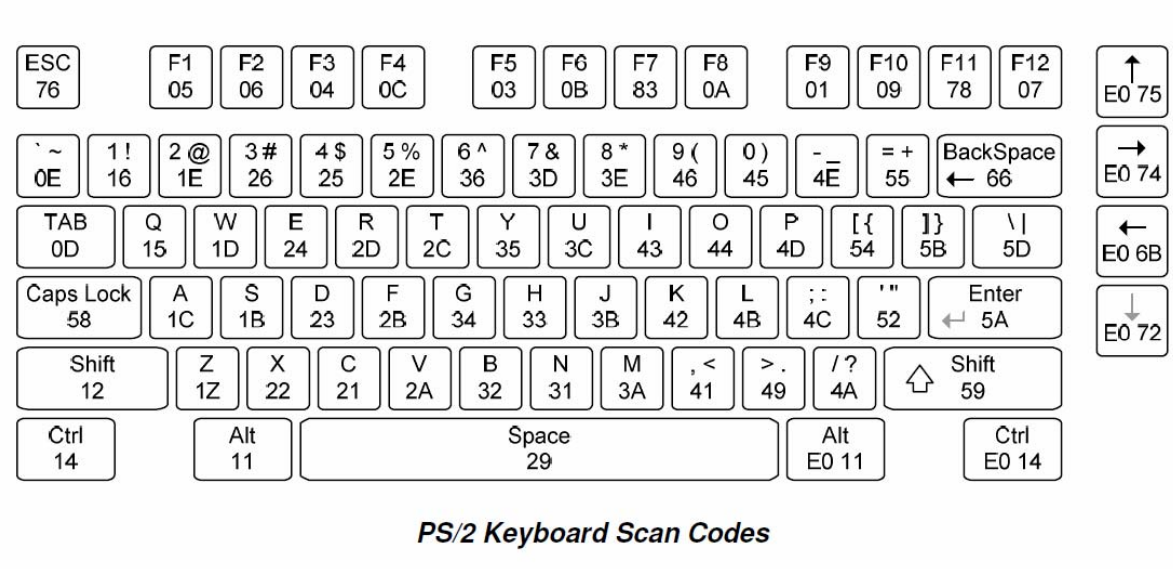


Рисунок П1.16 - Коды сканирования PS/2 клавиатуры

Клавиатура может отправить данные ведущему устройству только в том случае, когда обе линии (данных и синхронизации) находятся в высоком состоянии (или неактивны). Так как ведущее устройство - устройство управления шиной, то, перед отправкой данных, клавиатура должна проверить, не занята ли линия. Чтобы упростить этот процесс, линия синхронизации используется в качестве сигнал “свободен для

передачи”. Если ведущее устройство подтягивает линию синхронизации к низкому логическому уровню, клавиатура не должна отправлять данные, пока линия не вернется в исходное состояние. Клавиатура отправляет данные ведущему устройству в 11-разрядных словах, которые содержат: стартовый бит ‘0’, 8 бит кода сканирования (сначала младший бит), бит четности и стоповый бит ‘1’. При передаче данных клавиатура генерирует 11 тактовых импульсов с частотой от 20 до 30 кГц. Данные доступны для чтения по отрицательному фронту тактового импульса.

Подключение манипулятора «мышь»

При перемещении мышь изменяет значения линий данных и синхронизации, в состоянии покоя эти линии остаются в состоянии логической ‘1’. При каждом перемещении мыши, на ведущее устройство передаются три 11-разрядных слова. Каждое из этих слов содержит стартовый: бит ‘0’, 8 бит данных (сначала младший бит), бит четности и стоповый бит ‘1’. Таким образом, каждая передача данных содержит 33 бита, где биты с номерами 0, 11 и 22 ‘нулевые’ - стартовые биты, а биты с номерами 10, 21, и 33 ‘единичные’ - стоповые биты. Три 8-разрядных поля данных содержат информацию о перемещении, как показано на рисунке П1.16. Данные доступны для чтения по отрицательному фронту тактового импульса с частотой 20 - 30 кГц.

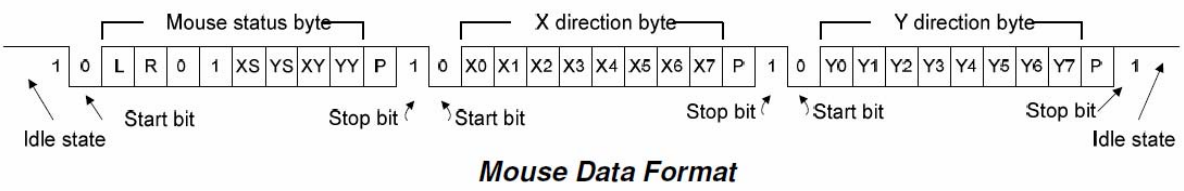


Рисунок П1.16 – Формат данных при работе с манипулятором «мышь».

Мышь предполагает использование относительной системы координат, в которой перемещение мыши направо генерирует положительное число в поле X, а перемещение налево генерирует отрицательное число. Аналогично, перемещение вверх мыши генерирует положительное число в поле Y, а перемещение вниз - отрицательное число (биты XS и YS в байте состояния - биты знака, где ‘1’ указывает на отрицательное число). Значения чисел X и Y представляют скорость перемещения мыши - чем больше число, тем быстрее мышь перемещается (биты XV и YV в байте состояния - индикаторы переполнения перемещения, где ‘1’, означает, что произошло переполнение). Если мышь перемещается непрерывно, 33-разрядные передачи повторяются каждые 50 мс. Поля L и R в байте состояния указывают на нажатие кнопки Left and Right (‘1’, указывает, что кнопка нажата).

Простые устройства ввода/вывода

Предусмотренные на плате Atlys ресурсы для ввода и вывода сигналов от контактов FPGA представлены на рисунке П1.17. На плате размещены содержит шесть кнопок, восемь переключателей, и восемь светодиодов. Одна из шести кнопок на печатной плате имеет маркировку "reset". Эта кнопка не отличается от пяти других, но может быть использована в качестве сброса в процессорных системах. Кнопки и

переключатели соединяются с FPGA через резисторы, чтобы предотвратить повреждение ПЛИС от случайных коротких замыканий. Аноды светодиодов соединяются с FPGA через резисторы на 390 Ом, и ярко светятся при протекании тока в 1мА.

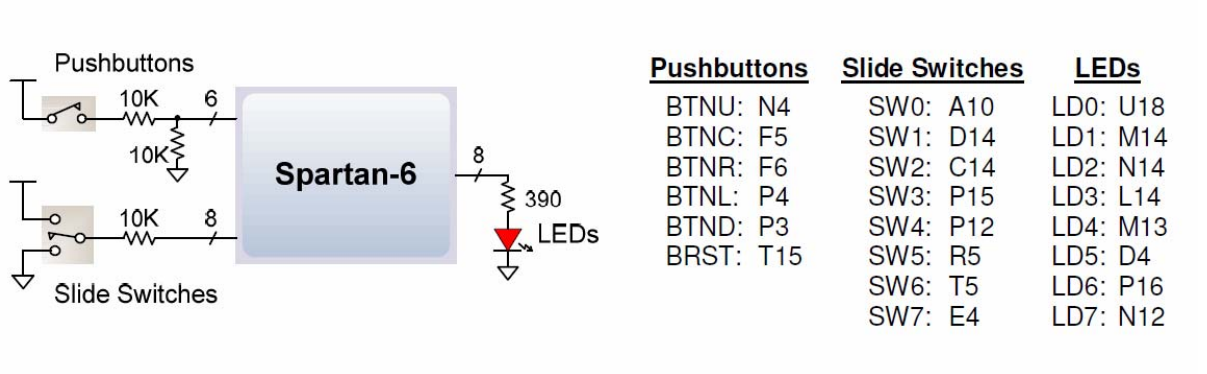


Рисунок П1.17 – Кнопки, движковые переключатели и светоизлучающие индикаторы платы ALTYS.

Разъемы для подключения внешних устройств

Плата Atlys содержит 68-контакный VHDC разъем для высокоскоростной параллельной передачи данных и 8-pin Pmod разъем для низкоскоростной передачи данных.

Разъем VHDC схема подключения которого приведена на рисунке П1.18, включает 40 линий данных (размещенных как 20 управляемых импедансом дифференциальных пар), 20 линий земли (по одной на пару), и восемь линий питания. Этот разъем, обычно используемый в SCSI 3 устройствах, может поддерживать скорости передачи данных до нескольких сотен мегагерц на каждом контакте.

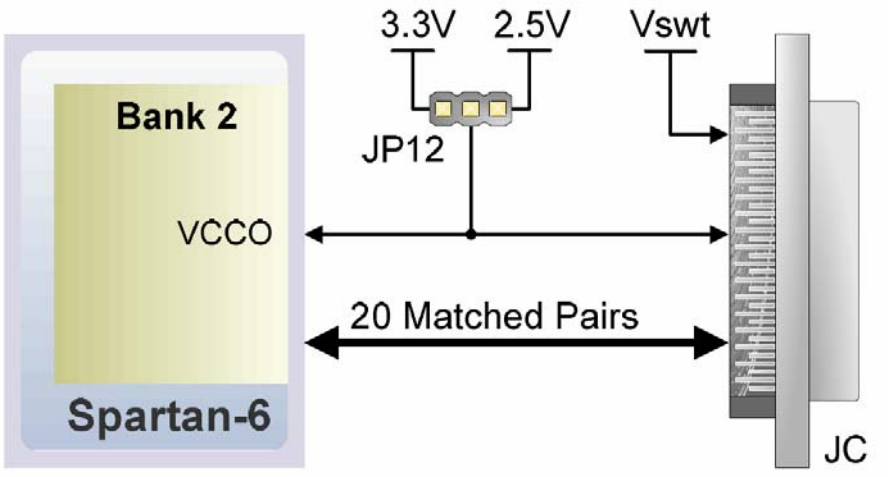
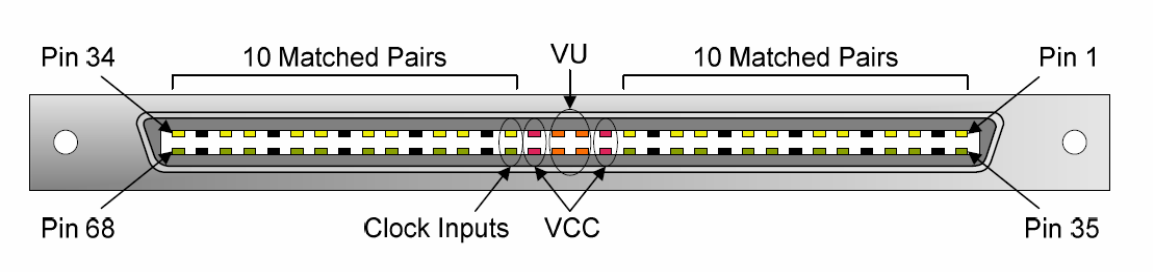


Рисунок П1.18 – Схема подключения разъема VHDC.

Все контакты FPGA, подключенные к разъему VHDC, располагаются во втором банке ввода-вывода. Контакты питания ячеек ввода-вывода банка 2 и четыре контакта Vcc разъема VHDC соединяются с отдельным слоем печатной платы, который может быть соединен с источником питания 2.5V или 3.3V, в зависимости от позиции перемычки JP12. Такое соединение позволяет периферийным платам и FPGA совместно использовать один источник питания Vcc и одинаковое напряжение уровней сигналов.

Нерегулируемый источник питания Vswt (номинально 5V) подключен к четырем другим контактам VHDC и позволяет передавать дополнительно ток до 1A к периферийным платам.

Все контакты ввода-вывода разъема VHDC смонтированы как согласованные пары, чтобы поддерживать LVDS (дифференциальный режим) передачу сигналов. Разъем использует симметричную расположение контактов по вертикальной оси, чтобы можно было подключать как периферийные платы, так и другие системные платы. Контакты 15 и 49 разъема соединены с входами синхронизации FPGA.



VHDC Connector Pinout

IO1-P:	U16	IO1-N:	V16	IO11-P:	U10	IO11-N:	V10
IO2-P:	U15	IO2-N:	V15	IO12-P:	R8	IO12-N:	T8
IO3-P:	U13	IO3-N:	V13	IO13-P:	M8	IO13-N:	N8
IO4-P:	M11	IO4-N:	N11	IO14-P:	U8	IO14-N:	V8
IO5-P:	R11	IO5-N:	T11	IO15-P:	U7	IO15-N:	V7
IO6-P:	T12	IO6-N:	V12	IO16-P:	N7	IO16-N:	P8
IO7-P:	N10	IO7-N:	P11	IO17-P:	T6	IO17-N:	V6
IO8-P:	M10	IO8-N:	N9	IO18-P:	R7	IO18-N:	T7
IO9-P:	U11	IO9-N:	V11	IO19-P:	N6	IO19-N:	P7
IO10-P:	R10	IO10-N:	T10	IO20-P:	U5	IO20-N:	V5

Рисунок П1.19 – Конструкция и назначение контактов разъема 68-pin VHDC.

Разъем Pmod 2x6 содержит: две линии VCC (контакты 6 и 12), две линии земли (контакты 5 и 11), и восемь сигнальных линий. Линии VCC и земли могут потреблять ток до 1A. Перемычка JP12 позволяет выбрать напряжение Pmod Vcc 3.3V или 2.5V. Линии данных Pmod не являются согласованными парами и не обеспечивают согласования импеданса и задержек.

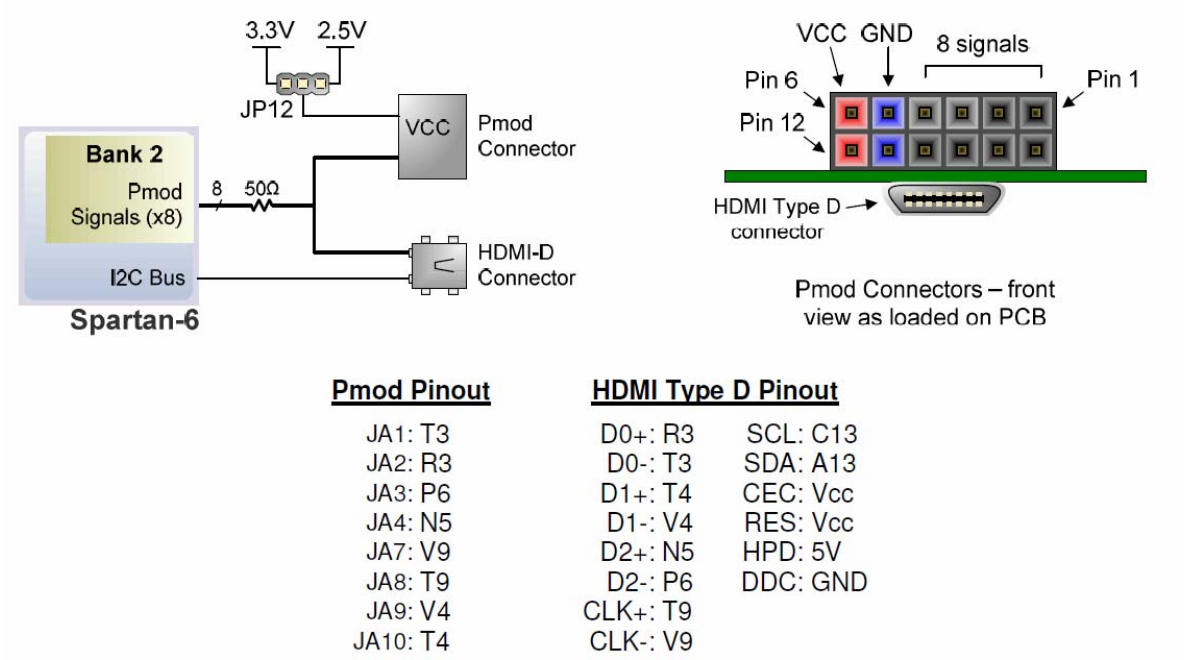


Рисунок П1.20 – Схема подключения, конструкция и контакты разъема Pmod 2x6.

Восемь сигнальных линий Pmod используются совместно с восемью сигналами данных, разъема D HDMI. Разъем HDMI, расположенный под разъемом Pmod с обратной стороны платы, имеет шину I2C и соответствует спецификации распиновки типа D HDMI, таким образом, он может использоваться в качестве второго выходного порта HDMI.

Самотестирование платы ALTYS

Демонстрационная конфигурация загружается в SPI Flash ROM платы Atlys во время изготовления. Этот демонстрационный пример, также доступный на веб-сайте Digilent, может играть роль теста, так как он взаимодействует со всеми устройствами и портами на плате. Если демонстрационный проект присутствует в SPI Flash при включении Atlys, сначала тестируется DDR, затем файл изображения передается от SPI Flash в DDR2. Это изображение выводится на HDMI J2 порт. Переключатели соединяются со светодиодами. Кнопки BTNU, BTND, BTNR, BTNL, BTNC, и RESET позволяют изменять частоты синусоидальных сигналов, поступающих на аудиовыходы LINE OUT and HP OUT.

Если тестовый проект не находится в SPI Flash ROM, он может быть загружен в FPGA или SPI Flash ROM при помощи программного обеспечения Adept.

Все платы Atlys проходят полное тестирование после изготовления. Если какое-либо устройство на плате Atlys не проходит тест или не отвечает должным образом, вероятно, что повреждение произошло во время транспортировки или использования. Типичными повреждениями являются напряженные паяные соединения и загрязнение в переключателях и кнопках, приводящих к неустойчивым отказам. Напряженные паяные соединения могут быть восстановлены повторной пайкой, припой и грязь могут быть убраны чистящими средствами для электроники. Если плата не прошла тест, и гарантийный срок не истек, то она будет заменена бесплатно.